공학박사학위논문

맵리듀스에서의 병렬 조인을 위한 다차원 범위 분할 기법

Multi-Dimensional Range Partitioning for Parallel Joins in MapReduce

2014년 8월

서울대학교 대학원
전기컴퓨터공학부
명 재 석

Abstract

Multi-Dimensional Range Partitioning for Parallel Joins in MapReduce

Jaeseok Myung
School of Computer Science & Engineering
The Graduate School
Seoul National University

Joins are fundamental operations for many data analysis tasks, but are not directly supported by the MapReduce framework. This is because 1) the framework is basically designed to process a single input data set, and 2) MapReduce's key-equality based data grouping method makes it difficult to support complex join conditions. As a result, a large number of MapReduce-based join algorithms have been proposed.

As in traditional shared-nothing systems, one of the major issues in join algorithms using MapReduce is handling of data skew. We propose a new skew handling method, called Multi-Dimensional Range Partitioning (MDRP), and show that the proposed method outperforms traditional skew handling methods: range-based and randomized methods. Specifically, the proposed method has the following advantages: 1) Compared to the range-based method, it considers the number of output tuples at each machine,

which leads better handling of join product skew. 2) Compared with the randomized method, it exploits given join conditions before the actual join begins, so that unnecessary input duplication can be reduced.

The MDRP method can be used to support advanced join operations such as theta-joins and multi-way joins. With extensive experiments using real and synthetic data sets, we evaluate the effectiveness of the proposed algorithm.

Keywords : Parallel Join, Data Skew, Multi-Dimensional Range Partitioning, MapReduce

Student Number : 2007-20973

Contents

List of Figures

List of Tables

Chapter I

Introduction

Enterprises today collect vast amounts of data from different sources and are expected to process them efficiently in order to provide new functionalities and services to users. To handle the unprecedented amounts of data, shared-nothing systems have received attention due to its scale-out features. A representative example is the MapReduce framework [1]. After the Apache Hadoop [2] community released an open-source MapReduce implementation, the parallel data processing framework has been widely used due to its salient features including scalability, fault-tolerance, and ease of programming.

This dissertation is about handling data skew in join algorithms using MapReduce. As in traditional shared-nothing systems, the processing time of a MapReduce-based join algorithm depends on the completion time of its longest running tasks. In this situation, skew handling methods are surely important to MapReduce-based join algorithms. As a result, we propose a new skew handling method that is applicable to a number of applications involved in parallel joins. The proposed method is compared to traditional skew handling methods: range-based and randomized methods. Throughout the dissertation, we show strong and weak points of the proposed method with illustrative examples in real-world applications.

1.1 Motivation

Joins are fundamental operations for many data analysis tasks, but are not directly supported by MapReduce [1]. The framework is basically designed to process a single input data set, and MapReduce's key-equality based data grouping method makes it difficult to support complex join conditions. Although a large number of MapReduce-based join algorithms have been proposed so far, researches on this problem are still actively conducted: recent studies [3, 4] provide insightful summaries on this problem.

One of the major issues in join processing with MapReduce is handling of data skew. As in traditional shared-nothing systems, the processing time of a join operation depends on the completion time of its longest running tasks. If the underlying data is sufficiently skewed, any of the gains from parallelism will be lost.

Unfortunately, most of MapReduce-based join algorithms exploit hash-based partitioning approach. A main reason is that Hadoop —the most popular MapReduce implementation—uses hash partitioning as a default option [2]. However, the basic approach does not consider skew inherent in the data sets. Repeated values in a join attribute are representative examples of data skew. Input tuples having the same join attribute value will be partitioned into the same reducer, which leads to significant imbalances across machines. Therefore, join algorithms based on the hash partitioning, such as repartition join [5], will show unacceptable performance at the presence of data skew.

An alternative approach to the hash partitioning is the range-based par-

titioning approach [6]. In this approach, each sub-range of join attribute values is assigned to a reducer. Since all sub-ranges are expected to have the same number of input tuples, some join attribute values can be overlapped across multiple sub-ranges. Repeated values are accordingly divided into several reducers. An attractive aspect of the range-based approach is that it is relatively easy to determine *approximate* sub-ranges via sampling. As shown in [6], a small number of samples are still helpful to determine boundaries of buckets. Thus, with a small cost for sampling, the range-based approach is effective regardless of the actual distribution of join attribute values. As a result, the range partitioning has been widely used in traditional shared-nothing systems and has been recently adopted for MapReduce [7].

Another alternative is the randomized partitioning approach [8]. It exploits a cross-product space between two input relations S and T. A row and a column represent individual input tuples from both relations. The cross-product space is covered by k rectangles, each of which represents one reducer. The areas covered by each rectangle should be the same so that we can expect the same number of input tuples for each reducer. We now assume an input tuple s from a relation S. For a given input tuple s, a random row s' is selected by the algorithm. Then, all reducers intersecting the s' row will receive the original input tuple s. Another input tuple t corresponding to a certain column will be processed similarly. Eventually, s and t meet at a certain reducer R, and a join result can be produced if s and t satisfy given join conditions. Due to the random selection, there are some rows and columns that are selected for multiple input tuples, while others are not selected at all. However, the large data size prevents significant variations

among rows and columns. This idea is applicable to arbitrary join conditions and several join types.

In this dissertation, we present a new skew handling approach to efficient processing of parallel joins in MapReduce. Since our approach extends the range partitioning, we call it as Multi-Dimensional Range Partitioning (MDRP). The proposed approach is designed to overcome limitations of traditional approaches.

A limitation of the original range-based approach is that the size of join results is ignored when it determines the sub-ranges. This is because the range partitioning only exploits samples from the most skewed relation. Joins are binary operations which two relations are participated. Without information of both relations, a serious imbalance can arise, such as join product skew[1]. Therefore, the basic idea underlying our improvement is a creation of a partitioning matrix instead of a one-dimensional partitioning vector. In our matrix, a dimension represents a relation to be joined. Then, a cell in the partitioning matrix forms a sub-range considering both relations. Using samples from both relations, we can estimate workloads of the cell in terms of both input and output tuples without requiring whole scan of data. If a cell has too heavy workloads to process, we can chop the heavy cell in order to balance the workloads.

The randomized approach also has a limitation. In the approach, one cannot specify a reducer in which an output tuple is actually produced. For correctness of join results, all input tuples have to be duplicated for all reduc-

[1]The join selectivity on individual nodes may differ, leading to an imbalance in the number of output tuples produced

ers intersecting with the randomly selected rows and columns. This random selection provides us a strong guarantee of load balancing, but it also incurs high input duplication. As the size of input relation increases, the amount of input duplication is also increased. In addition, as more reducers are employed, more reducers intersect with a random row or column, which leads more input duplication. In contrast, the MDRP approach assigns deterministic sub-ranges to a reducer. An input tuple is delivered to a reducer when its sub-range satisfies given join conditions. This helps us to avoid unnecessary input duplication. For example, in equi-join cases, our approach does not have to make a copy of an input tuple.

1.2 Contribution

We present a new skew handling technique, named Multi-Dimensional Range Partitioning (MDRP), that provides several benefits:

- **Efficiency:** The proposed technique is more efficient than previous skew handling techniques, range-based and randomized partitioning techniques. When a join operation has join product skew, our algorithm outperforms the range-based algorithms. When the size of input relation is sufficiently large, our algorithm outperforms the randomized algorithms.

- **Scalability:** Regardless of the size of input data, we can create sub-ranges that can be fit in memory. Moreover, the execution time of a join operation can be reduced as we add more machines into the

cluster. On the other hand, the randomized algorithm produces more intermediate results when we increase the number of processing units.

- **Platform-Independence:** Although we only examine our approach with the MapReduce framework, the MDRP technique itself can actually work with traditional parallel DBMSs. The range-based approach already used in many shared-nothing systems. Our algorithm improves the original range-based approach when we need to consider join results.

- **Advanced Join Processing:** It is applicable to several join types such as theta-joins and multi-way joins. We discuss several issues on extending our technique to handle other join types.

- **Ease to Use:** The implementation of our algorithms does not require any modification on the original MapReduce environment. Moreover, our technique can be embedded to other join algorithms, such as repartition join [5].

Our main contributions are listed as follows:

- We propose a new skew handling technique that extends current range-based approach. Benefits from our approach are described above.

- We provide implementation details of the approach with intuitive examples on the Hadoop's MapReduce environment. The efficiency of our approach is evaluated with extensive experiments.

- We investigate the effectiveness of our approach over applications in

practice. In graph pattern matching and matrix multiplication problems, our approach improves the performance of current join algorithms.

1.3 Outline

The rest of the dissertation is organized as follows. We briefly review the MapReduce framework and representative MapReduce-based join algorithms in Chapter 2. Then, we present our MDRP approach and discuss the effectiveness of our approach in Chapter 3. Subsequently, we extend the technique in order to process joins among multiple relations in Chapter 4. Especially, in Chapter 5, we investigate several applications that can be benefit from our techniques. Finally, Chapter 6 concludes the dissertation.

Chapter II

Backgrounds and Related Work

This chapter describes the background knowledge about join processing in MapReduce. We briefly review the MapReduce framework in Section 2.1, and featured join algorithms using MapReduce are introduced in Section 2.2. More importantly, we elaborate the problem of data skew in join processing in Section 2.3 and describe current states-of-the-art skew handling approaches in Section 2.4.

2.1 MapReduce

MapReduce provides a simple programming model for large-scale data analysis tasks on a shared-nothing environment [1]. The programming model consists of two primitives, Map and Reduce:

map	`(k1,v1)`	$\rightarrow$	`list(k2,v2)`
reduce	`(k2,list(v2))`	$\mapsto$	`list(k3,v3)`

The Map function is applied to an individual input record in order to produce a list of intermediate key/value pairs. The Reduce function receives a list of all values having the same *key* and produces a list of new output key/value pairs.

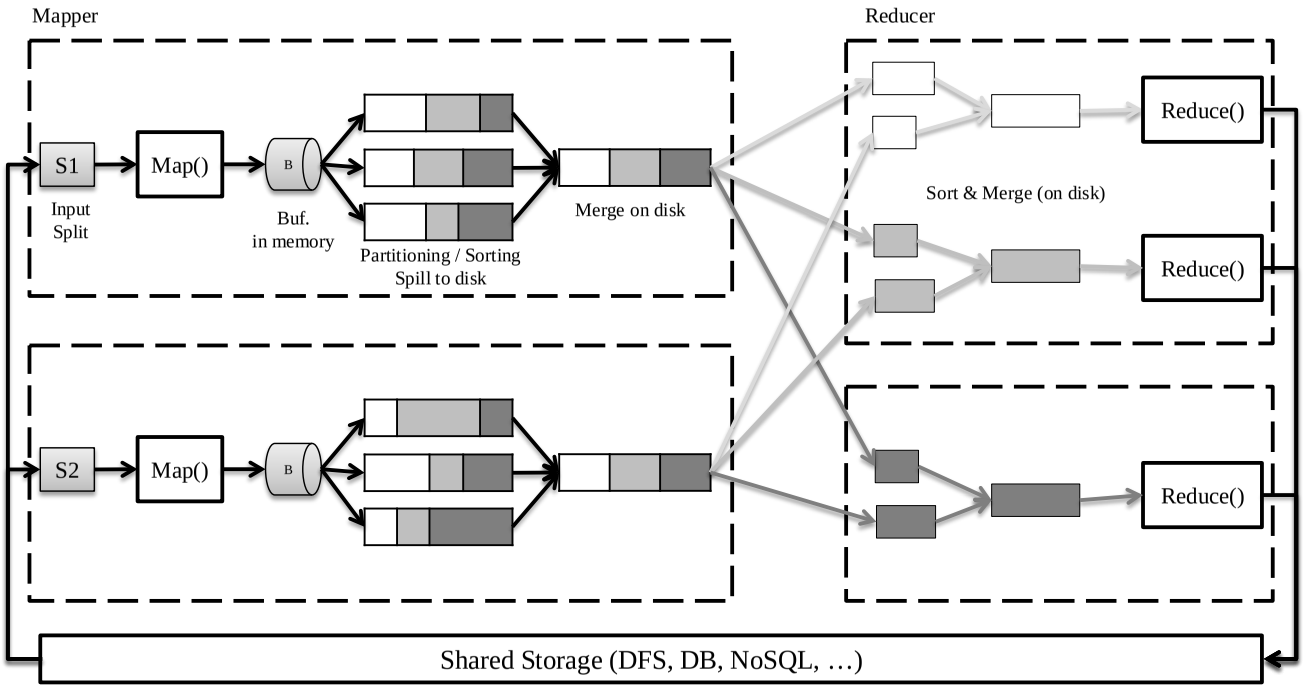


Figure 1: MapReduce overview

Figure 1 show the processing steps of a MapReduce job. Generally, a shared storage, such as HDFS, RDBMSs, NoSQL databases, is employed in order to store data. When a job is submitted to JobTracker, the data is split to different mappers and is processed by the Map function. The Map function receives an input tuple at a time and produces intermediate key-value pairs. The intermediate pairs are initially stored in a memory buffer, but they are spilled into local file systems when the buffer is filled with intermediate data.

It is notable that the intermediate key-value pairs are partitioned and sorted according to the key when they are spilled into the local disk. The spill files are in the end merged into one sorted file. In Hadoop's MapReduce, an intermediate pair is basically partitioned by the hash function, i.e. (partition_id=key mod # reducers). Therefore, the system has several partitions that are internally sorted by given *key* values.

The data from the Map function are shuffled, i.e., exchanged and merge-sorted, to reducers executing the Reduce function. It should be mentioned that the shuffle phase is a time-consuming task like the other map and reduce phases (due to copy of data thourgh network and merge-sort of individual partitions). Therefore, minimizing the size of intermediate key-value pairs is an important issue in programming a MapReduce job.

The Reduce function is invoked once for each distinct *key* and is applied on the set of associated values for that *key*, i.e., the pairs with same key will be processed as one group. As the mappers sort the intermediate key-values, input tuples of each Reduce function are guaranteed to be processed in increasing *key* order. Finally, the output of the Reduce function are

stored to the shared storage.

Despite its evident merits, MapReduce has some limitations when we want to implement a MapReduce-based join algorithm. First, the framework is originally designed in order to process a single input data. A typical MapReduce job is supposed to consume input from one file, thereby complicating the implementation of join operations. Second, MapReduce has a key-equality based data flow management scheme. When we want to implement a theta-join algorithm or a multi-way join algorithm, the data flow enforces us to make an inefficient algorithm.

2.2 Join Algorithms in MapReduce

In response to the limitations of MapReduce, a large and growing number of join algorithms have been proposed. Even though most of them do not consider data skew, it is valuable to review some featured join algorithms and discuss pros and cons of current algorithms.

2.2.1 Two-Way Join Algorithms

2.2.1.1 Map-Side Join vs. Reduce-Side Join

Join algorithms using MapReduce can be categorized into two classes: Map-Side Join algorithm (MSJ) and Reduce-Side Join algorithm (RSJ). Since the shuffle phase and the reduce phase require significant amount of communication and computation costs, MSJ algorithms are preferred in most cases. However, to execute a MSJ algorithm, the underlying data is stored and partitioned in a specific way according to requirements of the

join algorithm. On the other hand, the RSJ algorithms do not depend on the preprocessing of data. Therefore, in general cases, users prefer the RSJ algorithms because they do not have prior knowledge about data.

2.2.1.2 Repartition Join

The repartition join [5, 9] is a representative Reduce-Side Join algorithm. The basic idea is that mappers attach a tag to an input tuple. The tag represents a relation in which the input tuple is contained. According to the intermediate *key* values, input tuples having the same join *key* are delivered to the same reducer. Through the shuffle phase, the intermediate records are sorted by the key and secondarily sorted by tags. In the Reduce function, we can evaluate the join operation like traditional parallel hash join algorithms. We create a buffer for input tuples from a (build) relation. And then input tuples from the other (probe) relation are examined with tuples in the buffer whether they satisfy given join conditions or not.

Since the algorithm is a fundamental building block of all MapReduce-based join algorithms, let us explain more details on the repartition join. Suppose that we want to compute an equi-join between two relations S and T. Algorithm 1 to 3 show details of the implementation. Since we also provide an example (in the perspective of skew handling) in Example 1, more detailed explanations will be provided later.

Algorithm 1 Repartition join (Map)

Input : *key*, null
Input : *value*, a record from either S or T
jk ← extract the join attribute from *value*
tagged_input←add a tag of either S or T to *value*
composite_key←(jk, tag)
output (composite_key, tagged_input)

Algorithm 2 Repartition join (Partition)

Input : *key*, composite_key
return *key*.jk mod # of reducers

Algorithm 3 Repartition join (Reduce)

Input : *key*, composite_key
Input : *listValue*, records for *key*, first from S, then T
create a buffer B_S for S
for each record s of S in *listValue* **do**
  store s in B_S
end for
for each record t of T in *listValue* **do**
  for each record s in B_S **do**
    output (null, new_record(s,t))
  end for
end for

2.2.1.3 Directed Join

The directed join [5, 9] is classified into a Map-Side Join algorithm. A requirement of this algorithm is that both S and T are already partitioned on the join key before the algorithm begins. For example, let us suppose that we have two relations S and T which are pre-partitioned on the join key, i.e. $S = S_1 \cup S_2 \cup ... \cup S_m$, $T = T_1 \cup T_2 \cup ... \cup T_n$. It should be guaranteed that joins are evaluated only in the same partition. Then, a map-only job is sufficient to process the join operation. During the initialization of mappers, S_i is retrieved from a shared storage. We then make a hash table with tuples from S_i. In the Map function, a input tuple t from T_i is received. Then, we can probe the hash table with the join key extracted from t. For each input tuple s matched with t, a join result is produced and stored into the shared storage. Without the shuffle and the reduce phases, the total processing time can be saved significantly.

2.2.1.4 Broadcast Join

The broadcast join [5] is also a kind of Map-Side Join algorithms. The requirement for the broadcast join is that the size of a relation $|S|$ has to be very small so that it can fit in the memory. For instance, let S be a user table and T be a log table of all users. The size of $|S|$ is usually very small than $|T|$ and is likely fit in memory.

Instead of moving both S and T across the network, the broadcast join replicates the smaller table S to all machines. Then, each mapper receive the entire S table and is able to create a hash table of S. In the Map function, an

input tuple t is received and produces join results with s in the hash table. This can also be implemented as a map-only job.

2.2.1.5 Semi Join

The semi join algorithm [5] requires three separate MapReduce jobs for a join operation. This algorithm is neither Map-Side Join nor Reduce-Side Join algorithms. The basic idea underlying this algorithm is to avoid sending the input tuples in S that will not join with T. If the join selectivity is low, it reduces the overall communication cost in MapReduce.

Specifically, in the first MapReduce job, unique join keys in S are extracted and stored in a single file $S.uk$. The unique join keys are used in the next job to filter referenced input tuples in T. With first two phases, we know which tuples are actually participated in the join operation. Therefore, in the third phase, we can avoid sending unnecessary tuples through networks. Finally, we can compute the actual join with all attributes in tuples.

2.2.1.6 Bloom Join

The Bloom join algorithm [10] is a kind of Reduce-Side Join algorithm. This algorithm exploits the Bloom filter [11] in order to filter input tuples that are not joined. Hence, this is similar with the semi join algorithm, but this algorithm provides a probabilistic solution. The Bloom filter is a probabilistic data structure used to test whether an element is a member of a set. It consists of an array of m bits and k independent hash functions. All bits in the array are initially set to 0. When an element is inserted into the array,

the element is hashed k times with k hash functions, and the positions in the array corresponding to the hash values are set to 1. To test membership of an element, if all bits of its k hash positions of the array are set to 1, we can conclude that the element is in the set. This decision may yield false positive, but false negatives are never produced.

Lee et al. [10] provides an implementation of Bloom join using a single MapReduce job. In the first Map phase (only related to an input relation and a part of mappers will be participated), the algorithm reads input tuples from S and create a local filter. The local filters are then merged into a single global filter. In the second Map phase (related to the other relation and remained mappers), the global filter is used to filter out input tuples from T. Therefore, if a tuple does not pass the filter, it would not be transferred to a reducer. Finally, in the Reduce function, join results are produced as in the other join algorithms.

2.2.1.7 Theta-Join Algorithms

Generally, theta-joins have not been considered in the MapReduce framework because of its key-equality based data flow management. Recently, [8] has shown that a randomized algorithm, called *1-Bucket-Theta*, can address this problem. In addition, the algorithm is good at skew handling because of its random selection. Since the algorithm is an important previous work, we will explain more details of the algorithm in Section 2.4.3. It should be mentioned that the algorithm is one of state-of-the-arts for skew handling and is extended by Zhang et al. [12] to handling multi-way join queries.

2.2.2 Multi-Way Join Algorithms

Although a number of studies on join algorithms concentrate on two-way equi-join operations, some featured algorithms have been proposed in order to address other join types. Especially, some multi-way join queries have been proposed in recent literatures [13, 14]. Since the MapReduce framework is designed to process a single input data, processing *n*-way operation is very challenging. The algorithms can be categorized into two classes: iteration-based and replication algorithms.

2.2.2.1 Iteration-Based Algorithms

A multi-way join operation can be simply processed as a sequence of two-way joins. However, a MapReduce job requires significant overhead for its initialization. Moreover, intermediate results between different jobs have to be stored in a shared storage. This yield unnecessary disk I/O over the network. Therefore, [14] has proposed an algorithm to combine multiple joins in a MapReduce job. According to key-equality of input relations, all relations are participated into a MapReduce job. If relations have the same join key, they form a group of relations to be joined within the job. The join key attribute is selected in a greedy manner, and multiple join key attributes can be selected for a single MapReduce job. Actually, even though relations do not have the same join key, they can be processed in the MapReduce job simultaneously because relations have their own tags. The Reduce function create several hash (build) tables and a probe input can be matched to a corresponding build input table. This allows to reduce the number of job

iterations. We will see more details of iteration-based multi-way join algorithms in Section 4.3.4.

2.2.2.2 Replication-Based Algorithms

In [13], Afrati et al. have proposed a join algorithm with a single MapReduce job. This algorithm create a virtual space among join key attributes. Each join key represents a dimension of the space. After that, the space is covered with reducers. In the Map function, an input tuple is received and we extract the join key attribute in it. With a hash function h on the join key, we can determine a dimension of the input tuple. However, we cannot know the other dimensions. Therefore, to generate correct join results, we duplicate the input tuple for all possible hash function values of other dimensions. This requires a number of input duplication. If the duplication factor is very large, the shuffle phase becomes very slow. As a result, there is a trade-off between the iteration-based algorithm and replication-based algorithm. We will explain more details on the trade-off in Section 4.3.4.

2.3 Data Skew in Join Algorithms

Although a number of MapReduce-based join algorithms have been proposed so far, only a few algorithms consider data skew in processing of parallel joins. However, traditional shared-nothing systems already suffered from the skew handling problem. In this section, we review the data skew problem studied in traditional shared-nothing systems.

Walton et al. [15] describes four types of data skew: tuple placement

skew (TPS), selectivity skew (SS), redistribution skew (RS) and join product skew (JPS). TPS occurs when different amounts of input tuples are stored among machines. In a parallel DBMSs tuple placement is also a responsibility of the system, so this is an important problem for administrators. Next, SS occurs when the selectivity of selection predicates varies between machines. RS is seen when there is a mismatch between a distribution of a join attribute and a redistribution mechanism (e.g. poorly designed hash functions). Finally, JPS is related to the difference of join selectivity at each machine.

Figure 2 shows a taxonomy of data skew in parallel joins. We also highlight the effects of data skew in terms of MapReduce. In parallel DBMSs, TPS can be solved, before join processing, by a good hash function and proper choices of partitioning column. If the data set is well partitioned according to a join attribute, TPS will not occur. In MapReduce, TPS is also not a problem because the Namenode coordinates entire data sets so that tuples are evenly spread across Datanodes. Next, SS becomes a problem only when it causes RS or JPS. In MapReduce, the selection of input tuples is performed by mappers which receive entire input data sets. Therefore, regardless of a join operation's selectivity, the size of input tuples for each Map function is the same. On the other hand, the size of input tuples of the Reduce function can differ, but this can be seen as RS skew.

Skew Type	Occurs When	In Parallel Join Algs.	In MapReduce (Hadoop)
Tuple Placement Skew (TPS)	The initial distribution of tuples varies between nodes	Not a problem It can be avoided with a good hash function and proper choices of partitioning column	Not a problem The Namenode coordinates data so that tuples are evenly spread across Datanodes
Selectivity Skew (SS)	The selectivity of selection predicates varies between nodes (e.g. range selection in range partitioning)	Not a problem It becomes a problem only when it causes RS or JPS	Not a problem The selection can be performed before the join so that all Mappers receive the same number of tuples
Redistribution Skew (RS)	There is a mismatch between a join key distribution and a redistribution mechanism	Can be a problem Most of studies focus on this problem with sampling and cost estimation techniques	Can be a problem In hadoop, the default partitioning method uses a simple hash function
Join Product Skew (JPS)	The join selectivity at each node differs, leading to an imbalance in the number of output tuples produced	Can be a problem There is no way to avoid the workloads	Can be a problem The performance depends on the slowest Mapper and Reducer

Figure 2: A taxonomy of data skew in parallel joins

However, RS and JPS can cause serious problems in both parallel DBMSs and MapReduce. In fact, most of studies (about skew handling in parallel DBMSs) focus on the RS problem with sampling and cost estimation techniques. Likewise, in Hadoop, the default partitioning method is a simple hash partitioning method, leading a serious RS problem when there are repeated values in input relations. Moreover, the JPS problem cannot be avoided with a good hash function. The workloads for producing join results cannot be avoided so an efficient load balancing algorithm (in terms of the number of output tuples) is essential.

There have already been a number of skew handling algorithms in parallel DBMSs. For example, Kitsuregawa and Ogawa [16], Hua and Lee [17] proposed efficient parallel hash join algorithms. However, a limitation of these algorithms is that they require the relations to be joined are completely scanned before the join begins. In terms of MapReduce, they require two separate and expensive jobs. Thus, these algorithms may perform much worse than the basic repartition join when the relations are not skewed. This is the reason why the range-based and randomized approaches are widely used in practice.

Although most of skew handling algorithms estimate the workloads before the actual join evaluation, there are some algorithms that handle data skew dynamically [18, 19]. In other words, they monitor workloads of each machine at run time. However, the monitoring task can also be a burden to systems in zero skew relations. Our approach is static but effective regardless of the presence of data skew.

2.4 Skew Handling Approaches in MapReduce

Handling data skew is an important issue in MapReduce. TopCluster [20] and SkewTune [21] are representative researches, but they only consider a single input data set, which is not applicable to join operations. Therefore, a MapReduce-based implementation [7] of the range-based approach can be seen as the first skew handling join algorithm in MapReduce. Around the same time, Okcan and Riedewald [8] proposed the randomized approach which is applicable to any join conditions. Recently Zhang et al. [12] extended the randomized approach to multi-way theta-join queries. In this paper, we compared these state-of-the-art approaches with our proposed approach and discussed pros and cons of our approach.

2.4.1 Hash-Based Approach

A general and representative join algorithm using MapReduce is the repartition join [5] (a.k.a. reduce-side join [9]). The algorithm uses the hash-based redistribution method in order to balance workloads of join processing. But, the hash partitioning algorithms is unable to prevent data skew caused by repeated values. Let us consider a two-way equi-join example as illustrated in Figure 3:

Example 1 *(Repartition Join)* Relation S and T have eight data tuples respectively, and we have $k = 4$ machines in a cluster. The data schema is very simple, which contains three attribute fields (pk,jk,$others$) where the jk attribute is the join attribute. Input relations are split and stored in a distributed file system. Once the join begins, input tuples are fed to a Map function. Let

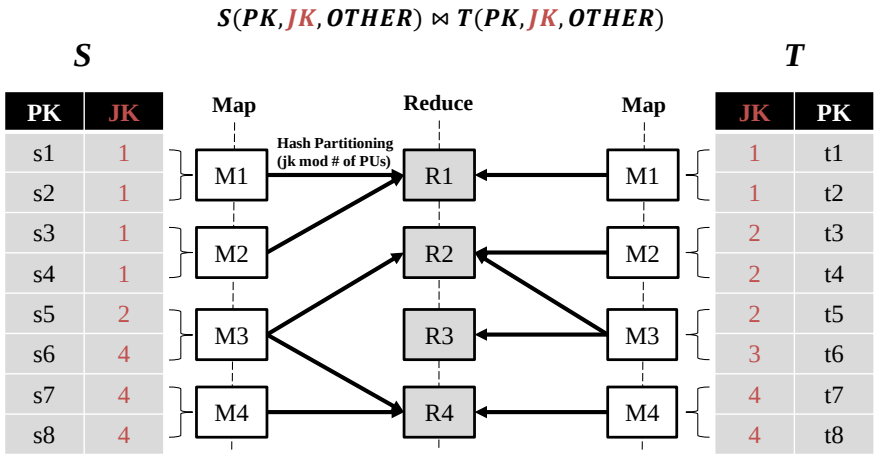


Figure 3: Hash-based partitioning approach

s be an input tuple from S. The Map function takes s and create a key-value $(s.jk, s)$ pair. Then, the intermediate key-value pair is delivered to a Reduce function in the $(s.jk \bmod k)$-th reducer. An input record t from the relation T is processed similarly. Eventually, the Reduce function receives a list of input tuples that have the same jk attribute value, and we can use any single-machine join algorithm in order to produce join results. □

In the repartition join, data skew can degrade the system performance seriously. With the hash-based partitioning, a reducer may receive too many input records than the other reducers. In our example, a reducer $R1$ receives four input tuples $\{s1, s2, s3, s4\}$ from S and two input tuples $\{t1, t2\}$ from T respectively. The $R1$ reducer has to produce eight output tuples, which is very heavy compared to other reducers. For example, $R3$ receives an input tuple $\{t6\}$ and produces zero output tuples. The completion time of a MapReduce job depends on the last finished reducer. Therefore, handling

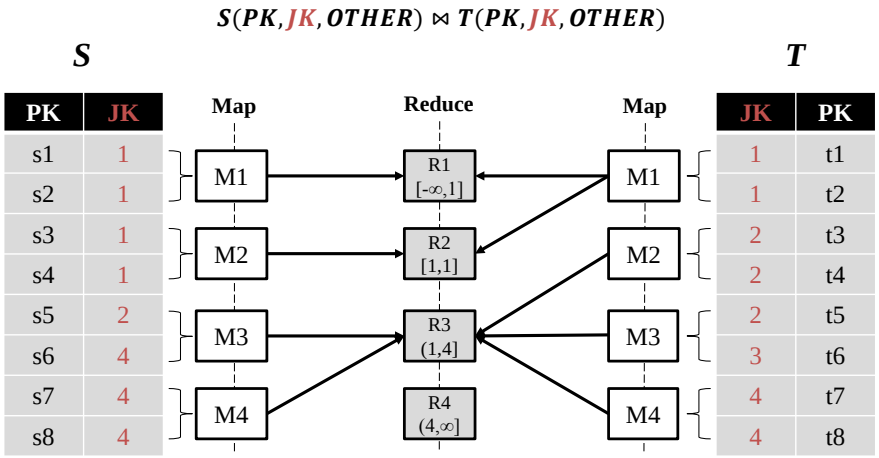


Figure 4: Range-based partitioning approach

data skew is very important in join algorithms in MapReduce.

2.4.2 Range-Based Approach

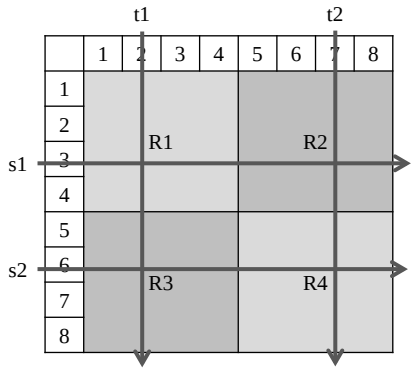
The range-based partitioning approach is a simple, but effective, solution to the skew handling problem [7, 6]. In the range-based approach, one first takes a pilot sample of both relations S and T. By counting the number of repeated samples in each, the most skewed relation can be determined (e.g., in Figure 4, the most skewed relation is S due to '1' values). Then, we can create a partitioning vector and evaluate the join operation as shown in the following example:

Example 2 *(Range Partitioning)* Let S' be a sorted list of samples from a relation S. In this example, for simplicity reasons, let us assume $S' = S$. Then, we have $S'.jk = \{1,1,1,1,2,4,4,4\}$. Since there are $k = 4$ reducers, we can select $k-1$ 'splitting values' which form the partitioning vector. The

$\lfloor |S'|/(k-1) \rfloor$-th elements are usual selections. In our example, as $\lfloor 8/3 \rfloor = 2$, the 2-th, 4-th and 6-th elements $\{1,1,4\}$ are selected. The partitioning vector $\{1,1,4\}$ creates 4 sub-ranges: $[-\infty,1]$, $[1,1]$, $(1,4]$ and $(4,\infty]$. Sub-ranges are assigned to different reducers. This contributes to balance the input workloads.

When the join begins, we can use the fragment-replicate technique [22]. Since $R1$ and $R2$ have sub-ranges $[-\infty,1]$ and $[1,1]$, input tuples, where $s.jk = 1$ or $t.jk = 1$, should be fragmented or replicated. To reduce the communication cost, a reasonable heuristic is to fragment the most skewed relation S and replicate the other relation T. Then, $\{s1,s2,s3,s4\}$ are divided into two reducers, and $\{t1,t2\}$ are duplicated. As a result, we have 2 reducers that produce 4 output tuples respectively instead of 1 reducer that produces 8 output tuples. Compared to the hash partitioning, the range partitioning allows an input tuple to be assigned two or more reducers. With the fragment-replicate technique, we can divide the expected output workloads. □

Our question about this range-based approach is that why do we discard samples from the less skewed relation, i.e. T. This can be a reason of serious imbalances. In Example 2, $R3$ has a sub-range $(1,4]$. Then, all data tuples, where $jk \neq 1$, are delivered to $R3$. This shows two possible problems: First, input tuples in the less skewed relation cannot be divided into the same size across all reducers; Second, join product skew cannot be even detected during the partitioning phase.



Map

Input tuple	Random row/col		Output
s1	3		(R1, s1), (R2, s1)
s2	6		(R3, s2), (R4, s2)
t1		2	(R1, t1), (R3, t1)
t2		7	(R2, t2), (R4, t2)

Reduce

Reducer	Input	Output
R1	s1, t1	(s1, t1)
R2	s1, t2	(s1, t2)
R3	s2, t1	(s2, t1)
R4	s2, t2	(s2, t2)

Figure 5: Randomized partitioning appraoch

Actually, DeWitt et al. [6] also proposed a technique, called virtual processor partitioning, to alleviate the join product skew problem. This technique assigns many virtual processors for each actual reducer so that we can create a longer partitioning vector than that of the original range partitioning. Since the longer partitioning vector indicates small size sub-ranges, highly sophisticated controls are enabled. However, a practical question still remains: how many virtual processors are needed? It is difficult to answer the question if we do not know the join selectivity. Obviously, we argue that exploiting samples from both relations can solve these problems.

2.4.3 Randomized Approach

A recent research [8] has shown that a randomized algorithm is effective to handle data skew. In addition, the algorithm works equally well for any join condition that belongs to $\{<, \leq, =, \geq, >, \neq\}$. We briefly review the

randomized algorithm with Figure 5 and the following example:

Example 3 *(Random Partitioning)* We first create a 8 by 8 matrix that represents the cross-product space between two relations S and T. The matrix can be covered by $k = 4$ reducers. To balance reducer's workloads, reducers should cover the same number of cells. Since $|S| = |T| = 8$, the best case is that a relation is divided to $\sqrt{k} = 2$ sub-ranges respectively. Therefore, the optimal partition is shown in Figure 5. Each reducer covers 16 out of 64 cells in total.

It is relatively easy to use the matrix in a join algorithm. For an input tuple s from S, the Map function finds all reducers intersecting the row corresponding to s in the matrix. A tuple t can be processed in a similar way. A reducer, that intersects with both s and t, is able to produce a join result. Since all reducers have the same number of cells, the number of input tuples are the same. However, the number of output tuples are likely different according to join selectivities for each reducer. This is the reason why the randomized approach is proposed.

The randomized approach chooses a random row s' for a given input row s and pretends as if s corresponds to the s' row. As shown in Figure 5, the $s1$ tuple is mapped to the row 3. Since the row 3 intersects with $R1$ and $R2$, the Map function creates two intermediate pairs $(R1, s1)$ and $(R2, s1)$. The input $s2$ randomly selects the row 6, and the Map creates $(R3, s2)$ and $(R4, s2)$. The random rows are represented as horizontal lines. They intersect with a number of vertical lines, each of which represents an input tuple t. Therefore, all combinations between s and t can be evaluated exactly once

across all reducers. This completes the join algorithm. □

Due to the random selection, there are some random rows and columns that are selected for multiple input tuples, while others are not selected at all. However, the large data size prevents significant variations among rows and columns. According to [8], the randomized algorithm practically guarantees that a reducer does not receive more than 1.1 times of its optimal input, and its output is not exceed 1.21 times its optimal size.

However, the randomized algorithm cannot avoid high input duplication. Every rows and columns from S and T are duplicated $\sqrt{k}$ times. As the size of an input relation becomes larger, the amount of duplication will also increase significantly. In addition, a large k also leads the high input duplication. This duplication always happens regardless of join conditions and a distribution of join attribute values. Conceptually, in a cross-product space, some cells can be filtered out according to the join condition (e.g. lower-left and upper-right corners in equi-join cases). We argue that our approach can remove non-candidate cells before the actual join begins because we use deterministic sub-ranges.

Chapter III

Our Approach

3.1 Multi-Dimensional Range Partitioning

We call our new skew handling approach as Multi Dimensional Range Partitioning (MDRP). Our partitioning approach outperforms current state-of-the-art approaches: the range partitioning and the random partitioning. In this chapter, we describe our MDRP approach.

3.1.1 Creation of a Partitioning Matrix

We first consider an equi-join of two relations S and T on a join attribute jk. We have k reducers, and the input relations can be partitioned into k sub-ranges: $\{S_1, S_2, ..., S_k\}$ and $\{T_1, T_2, ..., T_k\}$. For a relation S, the sub-ranges cover the *domain* of jk from α to β, $\alpha < S.jk \leq \beta$. Two special cases S_1 and S_k cover sub-ranges $[-\infty, \beta]$ and $(\alpha, \infty]$ respectively. The sub-ranges are sorted by their join attribute values. In other words, for all i and j, if $i > j$ then $S_i.\alpha \geq S_j.\alpha$ and $S_i.\beta \geq S_j.\beta$. Boundaries of sub-ranges can be determined by a partitioning vector as in the range partitioning. For example, let us consider two samples S' and T' from S and T relations: $S' = \{1,1,1,1,2,4,4,4\}$ and $T' = \{1,1,2,2,2,3,4,4\}$. As we have shown in Example 2, we can select $k-1 = 3$ splitting values $\{1,1,4\}$ and $\{1,2,3\}$, resulting $k = 4$ sub-ranges for each relation. As a result, we can create a

S

PK	JK
s1	1
s2	1
s3	1
s4	1
s5	2
s6	4
s7	4
s8	4

T

JK	PK
1	t1
1	t2
2	t3
2	t4
2	t5
3	t6
4	t7
4	t8

	T1 [∞-1]	T2 (1-2]	T3 (2-3]	T4 (3-∞]
S1 [∞-1]	8			
S2 (1-1]				
S3 (1-4]		3	0	6
S4 (4-∞]				0

Figure 6: An example of a partitioning matrix

partitioning matrix M as shown in Figure 6.

In the partitioning matrix M, i-th row represents S_i and j-th column is mapped to T_j. A cell (S_i, T_j) is classified into either a candidate cell or a non-candidate cell. Considering the given join condition, non-candidate cells will not produce any join results. In Figure 6, non-candidate (shaded) cells show the area of not satisfying the equi-join condition. For instance, (S_1, T_2) is mapped to sub-ranges $S : [-\infty, 1]$ and $T : (1, 2]$ which are not overlapped. Therefore, reducers do not have to cover non-candidate cells.

A candidate cell (S_i, T_j) has a value denoted by $M(i, j)$. The value indicates workloads of the cell that has to be processed. In our approach, we determine the workload $M(i, j) = |S'_i \bowtie T'_j|$ where S'_i and T'_j are samples in the sub-ranges. For example, $M(1, 1) = 8$ because we have 4 and 2 samples in S' and T' whose jk attribute values are '1'. Similarly, $M(3, 2) = 3$ because we have $\{s5\}$ and $\{t3, t4, t5\}$. The workloads of the other candidate cells can be computed in the same way.

Then, the partitioning matrix can be used for a load balancing algorithm. Before we start a MapReduce job, we can create a mapping between candidate cells and reducers. Reducers should be assigned the same number of cells and workloads as much as possible. When the Map function receives an input tuple s, we find candidate cells that contains $s.jk$ in its sub-ranges, and we obtain a list of reducers according to the mappings between cells and reducers. The load balancing algorithm has a responsibility to balance workloads across reducers.

An interesting aspect of the workload computation is that we do not consider the size of input tuples. The reason of this is every cell has almost the same size of input tuples. When we create a partitioning vector, we select splitting values from a *sorted* list of samples. Therefore, if the number of samples is large enough, every sub-range S_i has almost the same number of input tuples. We will discuss about the enough number of samples in Section 3.5.1.

It is also notable that $M(i, j) = 0$ should not be ignored for the actual join evaluation. Since we only exploit samples, some join results may not be produced when we create the partitioning matrix. For correctness of join results, all candidate cells have to be assigned to at least one reducer. In our example, reducers have to cover (S_3, T_3) and (S_4, T_4) cells.

3.1.2 Identifying and Chopping of Heavy Cells

In the matrix M, each cell has different workloads. A load balancing algorithm can be used for the same amount of workloads across reducers, but it is sometimes impossible to obtain an optimal mapping between cells

and reducers. We now define a heavy cell as follows:

Definition 1. (Heavy Cell) *A heavy cell in the partitioning matrix M is a cell* (S_i, T_j) *where*

$$\frac{M(i,j)}{\sum_{s=1}^{k}\sum_{t=1}^{k} M(s,t)} \geq \frac{1}{k}$$

is satisfied.

Intuitively, the ratio of heavy cell's workload to the total workloads is greater than or equal to $1/k$. Since, k is the number of reducers, $1/k$ indicates the optimal workload ratio for each reducer. Therefore, if there is a heavy cell, it is impossible to balance the workloads.

For example, in Figure 6, we have two heavy cells (S_1, T_1) and (S_3, T_4). The total workload is 17, and the optimal workload ratio is $1/4 = 0.25$. Since $M(1,1) = 8$, the (S_1, T_1) cell is a heavy cell $(8/17 \simeq 0.47 \geq 0.25)$. Similarly, (S_3, T_4) is also a heavy cell $(6/17 \simeq 0.35 \geq 0.25)$.

For the purpose of load balancing, the heavy cells have to be chopped. We now define ω as the optimal workload, i.e. $\omega = (\sum_{r=1}^{k}\sum_{s=1}^{k} M(r,s))/k$. In our example, $\omega = 17/4 = 4.25$. Then, heavy cells are divided into d cells until $M(i,j)/d \leq \omega$. For instance, (S_1, T_1) and (S_3, T_4) are chopped into 2 non-heavy cells because $8/2$ and $6/2$ are less than ω. By chopping a heavy cell, we can assign the heavy cell to two or more different reducers. This enables us to use the fragment-replicate technique. In Figure 7, we can see that heavy cells are divided into several non-heavy cells and assigned to different reducers.

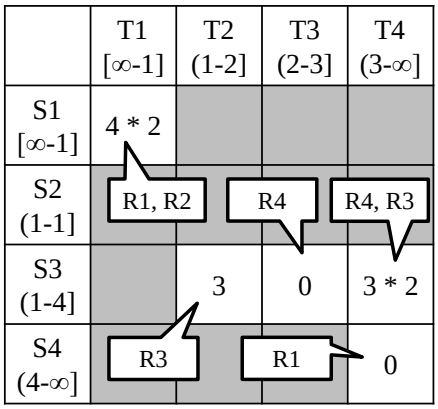


Figure 7: Mapping between cells and reducers

3.1.3 Assigning Cells to Reducers

We now have a set of non-heavy candidate cells $C = \{c_1, c_2, ..., c_{|C|}\}$ to be assigned to reducers. It should be clarified that c_l may differ from a cell (S_i, T_j) in the matrix M. The original (S_i, T_j) cell can be divided into several cells, and The C denotes a set of those non-heavy (chopped) cells. Thus, two or more cells c_l and c_m may refer the same cell (S_i, T_j). A cell $c_l \in C$ consists of sub-ranges S_i, T_j and its workload $w(c_l)$. In Example 4, we use $(S_i, T_j, w(c_l))$ in order to refer a cell c_l.

A load balancing algorithm assigns the cells to a set of reducers $R = \{R_1, R_2, ..., R_k\}$. Let C_i be a disjoint subset of C assigned to i-th reducer R_i. The number of cells in C_i is denoted $|C_i|$. We also use W_i in order to specify the total workloads assigned to the reducer, i.e. $W_i = \Sigma_{c_l \in C_i} w(c_l)$.

In our problem, the load balancing algorithm should satisfy the following constraints: First, all reducers are assigned the same number of cells as much as possible, i.e. $\lambda_{in} = (max(|C_i|)/avg(|C_i|)) \simeq 1$; Second, The total

workloads for each reducer should be the same as much as possible, i.e. $\lambda_{out} = (max(W_i)/avg(W_i)) \simeq 1$. Note that we have two criteria λ_{in} and λ_{out}. The first λ_{in} is about the number of input tuples, and the second λ_{out} is about the number of output tuples. Therefore, if we can satisfy both $\lambda_{in} = 1$ and $\lambda_{out} = 1$, the load balancing algorithm is the optimal algorithm. However, many load balancing algorithms, such as the greedy algorithm and the LPT algorithms [23], only consider a single constraint. Hence, we design a new load balancing algorithm that balances the input and output skew.

Algorithm 4 Assign

Input: $C = \{c_1, c_2, ..., c_{|C|}\}$, a set of non-heavy cells
Output: $\{C_1, C_2, ..., C_k\}$, mappings b/w cells and reducers

```
1: for each cell c_l in C in decreasing order of w(c_l) do
2:     R_i = getNextReducer()
3:     C_i = C_i ∪ {c_l}
4: end for
5: return C_1, C_2, ..., C_k
```

Algorithm 5 getNextReducer

Output: R_{idx}, a reducer to be assigned a cell

```
1: minCell = ∞, minLoad = ∞, idx = 0
2: for each reducer R_i do
3:     if minCell < |C_i| then
4:         minCell = |C_i|
5:     end if
6: end for
7: for each reducer R_i do
8:     if minCell = |C_i| and minLoad < W_i then
9:         idx = i
10:        minLoad = W_i
11:    end if
12: end for
13: return R_idx
```

In Algorithm 4 and 5, we show our load balancing algorithm. The algorithm is based on a greedy selection of reducers. We first sort cells in C according to their workload $w(c_l)$. For each cell c_l, we select a reducer R_i that has the minimum number of cells and workloads. Once R_i is determined by the `getNextReducer()` function, the cell c_l is added to C_i. We explain more details with Figure 7 and Example 4:

Example 4 *(Load Balancing Algorithm)* We have a set of non-heavy candidate cells C=$\{(S_1,T_1,4),(S_1,T_1,4),(S_3,T_2,3),(S_3,T_4,3),(S_3,T_4,3),(S_3,T_3,0),(S_4,T_4,0)\}$. Note that we have two $(S_1,T_1,4)$ and $(S_3,T_4,3)$ cells because the original cells are chopped into two cells respectively. In addition, cells are sorted according to their workload $w(c_l)$. Since, we have 4 reducers $\{R_1,R_2,R_3,R_4\}$. The set C will be divided into 4 subsets C_1,C_2,C_3,C_4. Initially, all $|C_i|=0$ and $W_i=0$. First, we take $(S_1,T_1,4)$ and a reducer R_1. Then, $C_1=\{(S_1,T_1,4)\}$, $|C_1|=1$ and $W_1=4$. Next, we receive another $(S_1,T_1,4)$, and a reducer R_2 is selected by the Algorithm 5. Then, $C_2=\{(S_1,T_1,4)\}$, $|C_2|=1$ and $W_2=4$. Similar process is applied to every $c_l \in C$. Finally, $C_1=\{(S_1,T_1,4),(S_4,T_4,0)\}$, $C_2=\{(S_1,T_1,4)\}$, $C_3=\{(S_3,T_2,3),(S_3,T_4,3)\}$, and $C_4=\{(S_3,T_4,3),(S_3,T_3,0)\}$. The $\lambda_{in}=(max(|C_i|)/avg(|C_i|))=2/1.75=1.14$ where $max(|C_1|)=2$, and $\lambda_{out}=(max(W_i)/avg(W_i))=6/4.25=1.41$ where $max(W_3)=6$. □

3.1.4 Join Processing using the Partitioning Matrix

Like the original range-based approach, we use the fragment-replicate technique for actual join processing. The mappings between cells and re-

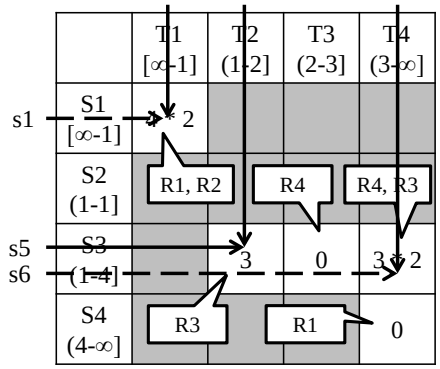


Figure 8: Join processing with a partitioning matrix

ducers $\{C_1, C_2, ...C_k\}$ are distributed and shared to all Map functions. Figure 8 shows how we use the mappings. Suppose that a Map function receives an input tuple $s1$ from S. Since $s1.jk = 1$, the input tuple belongs to a cell (S_1, T_1). The cell is mapped to two reducers $R1$ and $R2$. In other words, $(S_1, T_1, 4) \in C_1$ and $(S_1, T_1, 4) \in C_2$. As the tuple belongs to two different reducers, we now determine whether the input tuple should be fragmented or replicated. By counting the number of samples in S_1 and T_1 where $jk = 1$, we can know S has more repeated values than T. Therefore, we fragment the $s1$ input tuple; One of reducers is randomly selected and the tuple is delivered to the reducer. On the other hand, input tuples from T is replicated to both reducers. In Figure 8, dashed lines show the fragmentation of input data whereas solid lines represent the replication.

Before we start a MapReduce job, we create a partitioning matrix via sampling as described in Section 3.5.1. When the MapReduce job begins, we copy the partitioning matrix into all mappers and reducers. The DistributedCache [9] mechanism is employed in order to share the partitioning

matrix. Then, we can load the partitioning matrix from the setup() method of each mapper and reducer.

We now explain the implementation of the Map and the Reduce functions. Algorithm 6 shows the Map function. For simplicity reasons, the Map function receives an input tuple from S, but T is processed similarly. The Map function accesses the partitioning matrix which is loaded during the initialization of the mapper. M is a p by p partitioning matrix that contains candidate cells. In the $M.listCell$ function, we examine the input tuple $s.jk$ whether it satisfies the join condition with regard to cell's sub-range. Satisfied cells are returned by the $listCell$ function (line 1). Since we already mapped cells and reducers, we can easily obtain a list of reducers for each cell c in $listCell$ (line 2-3). The next step is to determine the tuple to be fragmented or replicated (line 4). As discussed in Section 3.1.4, it depends on the existence of more repeated values. If an input tuple s would be fragmented, we can select a random reducer and deliver the input tuple to a reducer (line 5-6). Otherwise, the input tuple is copied to all reducers that are assigned to a cell (line 8-10).

The Reduce function receives a list of input tuples from both relations S and T. Input tuples are sorted by the input relation, i.e. tuples from S are followed by tuples from T. This is enabled by the 'secondary sort' feature of the Hadoop MapReduce framework. We use the setGroupingComparatorClass() method in order to sort input tuples in the Reduce function. The input tuples from S are used as a build input whereas the input tuples from T are used as a probe input. Thus, we first create an empty set for the S relation and then add all S tuples into the set $listSTuple$ (line 1-4). After that,

Algorithm 6 Map

Input: an input tuple $s \in S$
Input: a partitioning matrix M
1: $listCell = M$.listCell$(s.jk, S)$
2: **for** each cell c in $listCell$ **do**
3: $R = c$.listReducer()
4: **if** M.fragment$(s.jk, S)$ == true **then**
5: i=random() % $|R|$
6: output ($R[i]$, (s, S))
7: **else**
8: **for** i=0 to $|R|$ **do**
9: output ($R[i]$, (s, S))
10: **end for**
11: **end if**
12: **end for**

for each combination of input s and t, we evaluate the join condition and produce an output tuple (line 5-9).

Algorithm 7 Reduce

Input: (reducerId, [(s_1, S),(s_2, S),...,(t_1, T),(t_2, T),...])
1: $listSTuple = \{\}$
2: **for** each (s_i,S) in input list **do**
3: $listStuple = listSTuple \cup \{s_i\}$
4: **end for**
5: **for** each (t_i,T) in input list **do**
6: **for** each s_j in $listSTuple$ **do**
7: output $s_i \bowtie t_j$
8: **end for**
9: **end for**

Note that, compared to the original range-based approach, our approach provides sophisticated fragment-replicate controls. The range-based approach determines an entire relation to be fragmented or replicated. However, our approach makes a decision *cell-by-cell*. For a cell, S can be fragmented,

while in the other cell S can be replicated. Generally, S and T have different distributions on the join attribute. Therefore, our approach can handle data skew in both relations.

Another aspect to be mentioned is that we do not duplicate an input tuple for all reducers intersecting with a row in the partitioning matrix. For example, the $s5$ and $s6$ tuples belong to a cell (S_3, T_2) and (S_3, T_4) respectively. We know that the other cells cannot make a join result with given input tuples. Therefore, we only consider reducers mapped to the cell. This enables us to reduce unnecessary communication costs, compared to the randomized approach.

3.2 Theoretical Analysis

The λ_{in} and λ_{out} show the effectiveness of a load balancing algorithm in terms of input and output skew. As shown in Example 4, the values are close to 1. In fact, it is impossible to exactly know the imbalance across reducers because our load balancing algorithm works with samples from entire data sets. However, assuming that we know the exact statistics about input relations, it is possible to roughly guess the upper bound. We analyze the worst case of our algorithm in terms of input and output skew.

Theorem 1. (Input Imbalance) *In the worst case, the input imbalance* λ_{in} *is less or equal to 2.*

Proof. We have $|C|$ cells to be assigned to different reducers. Since we always select a reducer which is the minimum number of cells assigned (greedy algorithm in Algorithm 4 and 5), a reducer is assigned either $\lfloor |C|/k \rfloor$

or $\lceil |C|/k \rceil$ cells. The difference between the maximum and the minimum number of cells is 1. Therefore, the worst case is that a reducer is assigned 2 cells whereas the others are assigned a single cell, i.e. $|C| = k+1, max(|C_i|) = 2$ and $avg(|C_i|) = (k+1)/k$. Then, the λ_{in} in the worst case is:

$$\lambda_{in} = \frac{max(|C_i|)}{avg(|C_i|)} = \frac{2}{(k+1)/k}$$

Since k is a positive integer, $\lambda_{in} \leq 2$. □

Intuitively, in the worst case, the input imbalance does not exceed twice than the optimal (average) case. Note that, the upper bound is still useful even if we do not know the exact statistics of input relations. The unit of load balancing is a cell (sub-range). Therefore, if we partition an input relation into equal sized partitions, the upper bound of load balancing is still valid. Next, we can analysis the output skew, λ_{out}:

Theorem 2. (Output Imbalance) *In the worst case, the output imbalance λ_{out} is less or equal to 2.*

Proof. Since we divided a heavy cell, the maximum ratio of a cell does not exceed $1/k$. We now consider a cell $c_1 \in C$ which has the largest $w(c_l)$ among non-heavy cells. Suppose that the workload $w(c_1)$ is $(1/k) - \varepsilon$, where ε is a small positive real number close to 0. Then, the worst case is that there was a heavy cell which was already chopped into k cells, i.e. $c_2, ...c_{k+1}$. The workload of the heavy cell was $\eta = 1 - w(c_1) = (1 - (1/k)) + \varepsilon$. Then, $w(c_2) = ... = w(c_{k+1}) = \eta/k$. While the optimal (average) workload is $1/k$,

the maximum workload for a reducer $max(W_i)$ is:

$$max(W_i) = w(c_1) + w(c_2) = (\frac{1}{k} - \varepsilon) + (\frac{(1-(1/k)) + \varepsilon}{k}) \\ \simeq (\frac{1}{k}) + (\frac{(1-(1/k))}{k}) = \frac{2k-1}{k^2}$$

Therefore, in the worst case, the output skew λ_{out} is:

$$\lambda_{out} = \frac{max(Wi)}{avg(Wi)} = \frac{(2k-1)/k^2}{1/k} = \frac{2k-1}{k}$$

Since k is a positive integer, $\lambda_{out} \leq 2$. □

In summary, the essence of our approach is as follows: First, compared with the range partitioning, we create a partitioning matrix instead of a partitioning vector. This enables us to consider the size of join results; Second, compared to the randomized algorithm, our approach exploits the partitioning matrix in order to filter out non-candidate (not satisfying given join conditions) cells before the actual join processing. This reduces unnecessary input duplication, leading lower communication cost.

3.3 Complex Join Conditions

Let us consider a general theta-join between two relations. Any theta-join is a subset of the cross-product. Since the partitioning matrix represents the cross-product space, any join condition can be considered. According to a given join condition θ, candidate cells and non-candidate cells are classified. Using samples from both relations, we can determine workloads of

each candidate cell.

In a theta-join case, an input tuple can belong to two or more cells. Therefore, input duplication can be increased compared to the equi-join cases. If the number of duplications is greater than $\sqrt{k}$, the randomized approach is more effective than our approach. However, in practice, many theta-join queries measure the distance of two tuples. For example, in a spatial database, we usually find nearest neighbors from a query point. In this case, the join condition θ is very selective and almost the same with equi-join queries. Therefore, input duplication is likely smaller than $\sqrt{k}$. In Section 3.4.3, we can see that our approach outperforms the current state-of-the-art theta-join algorithm.

We now consider a join operation among multiple relations that shares a single join attribute. This is a special case of multi-way joins. Actually, the MapReduce programming model is known as a good choice for processing multi-way joins. This is because the framework has the key-equality based data flow. Regardless of the number of input relations, input tuples that have the same join attribute value will be delivered to the same reducer. Then, we can use any single machine join algorithm to produce the join results. In our approach, we now consider a join among three relations R, S, T. They has the same join attribute JK. The partitioning matrix is then extended to a three-dimensional partitioning cube. A cell in the cube represents a sub-range considering three relations. Then, we can find heavy cells in the cube, and finally we can process the join operation similarly.

3.4 Experiments

To verify the effectiveness of our algorithm, we conduct experiments on a 13-machine Hadoop-1.1.2 cluster with both real and synthetic data sets. One machine served as the master node (Namenode and JobTracker), while the other 12 were the worker nodes (Datanode and TaskTracker). Every node has a single Intel (R) Core (TM) i7-3820 CPU 3.60GHz processor, 10MB cache, 8GB RAM, a 3.5TB hard disk. In total, the test bed has 48 cores with 8 GB memory per core available for Map and Reduce tasks. Each core is configured to run one map and one reduce task concurrently. All machines are directly connected to the same Gigabit network switch, and the distributed file system block size is set to 64MB. We run each experiment three times and report the average execution time.

In our experiments, we present results for following data sets:

Scalar Skew (xα): For a fixed α, we create a 100M tuple relation. The α is the number of tuples in which the value '1' appears in the join attribute (these tuples are chosen at random). The other tuples have a join attribute value chosen randomly from 2 to 100M. The data schema is very simple, which contains three attribute fields (pk, jk, $others$). The size of each tuple is 200 bytes. Thus, each relation occupies approximately 20GB (200 bytes * 100M) of disk space. The term 'scalar skew' is used in [6], and this data set helps us to understand exactly what experiment is being performed. This is also used in many skew handling related researches [7, 6, 18, 15].

Zipf Distribution (zβ): For a 100M tuple relation R with a domain of D distinct values, the i-th distinct join attribute value, for $1 \leq i \leq D$, has the

number of tuples given by the expression

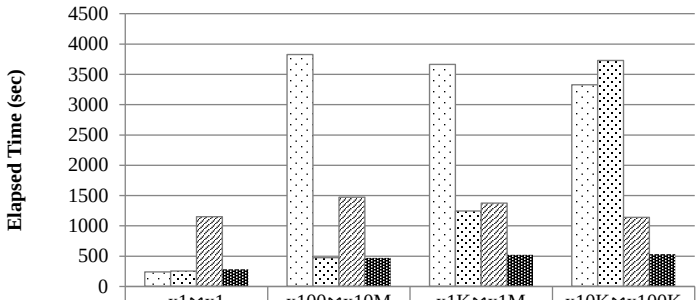
$$|D_i| = \frac{|R|}{i^{\beta} * \sum_{j=1}^{D} 1/j^{\beta}}$$

where β is the skew factor. When $\beta = 0$, the distribution becomes $uniform$. With $\beta = 1$, it corresponds to the pure Zipf distribution [24].

Cloud : This is a real data set containing extended cloud reports from ships and land stations [25]. There are 382 million records, each with 28 attributes, resulting in a total data size of 28.8GB. This data set is used by [8] in order to demonstrate the performance of the randomized algorithm. Thus, we selected this data set for a comparison purpose.

3.4.1 Scalar Skew Experiments

Figure 9 shows results for computing an equi-join on scalar skew data sets. We compare four partitioning approaches (HASH, RANGE, RANDOM and MDRP) with four different cases. For a fair comparison, we considered the followings: 1) The RANGE algorithm is an implementation of the virtual processor range partitioning [6] which is an improvement of the original range partitioning for handling join product skew. However, there is no further researches on the number of virtual processors. Therefore, we use 60 virtual processors for each reducers, which follows the best parameter in [6]. 2) The RANDOM algorithm is an implementation of the 1-Bucket-Theta algorithm proposed in [8]. The algorithm is the state-of-the-art theta-join algorithm in MapReduce. However, it requires a square matrix for a optimal mapping. As a result, in this experiment, we only use 36 reducers



	x1⋈x1	x100⋈x10M	x1K⋈x1M	x10K⋈x100K
HASH	240	3826	3663	3325
RANGE	252	472	1243	3727
RANDOM	1151	1470	1373	1139
MDR	281	468	521	532

Figure 9: Performance on scalar skew data sets

for all algorithms even though we have 48 cores in total.

The first case (x1 ⋈ x1) shows the zero skew case. We see that HASH shows the best performance among all algorithms. A main reason of this is that HASH does not incur the overhead of sampling, creating a partitioning matrix, and assigning cells to reducers. In addition, the Map function only need to compute a hash function in order to determine a destination reducer, while the other skew handling algorithms have to search a matrix for appropriate cells. However, compared to the HASH algorithm, RANGE and MDRP also show the similar elapsed time. Thus, we can see that the overhead of sampling is not significant. Considering gains from skewed test cases, this difference seems to be acceptable. Finally, the RANDOM algorithm is the worst choice because of its high input duplication.

The other cases (x100 ⋈ x10M, x1K ⋈ x1M, x10K ⋈ x100K) show results when join product skew exists. The number of output tuples is greater than 1 billion, and the output size is over 400GB. In this experiment, we

vary the degree of skew with keeping the total number of output tuples. Obviously, detecting join product skew from samples becomes more difficult from the second case to the last case. The HASH algorithm is a baseline which does not handle data skew.

In the second case, RANGE and MDRP algorithms are effective to handle data skew similarly. The RANGE algorithm uses samples from the x10M data set which leads a number of '1's in its partitioning vector. In the third case, the RANGE algorithm collect samples from x1M data set, which means less number of reducers produce output tuples whose join attribute values are '1'. As a result, longer elapsed time is required. This trend continues to the last case. In contrast, we can see that the MDRP algorithm shows consistent performance regardless of the degree of skew. It is notable that the RANDOM algorithm is also robust in the presence of data skew. However, its overhead of input duplication is significant, which leads longer elapsed time than that of our approach.

Table 1 explains more details on the experiment. In the table, each algorithm has three columns: In, Out and Time. The In and Out columns correspond to the ratio between the maximum and the optimum (average) number of input / output tuples respectively. The Time column represents the same ratio in terms of the processing time of reducers.

Table 1: Load Imbalance in Reducers (MAX / AVG)

	HASH			RANGE			RANDOM			MDRP		
	In	Out	Time	In	Out	Time	In	Out	Time	In	Out	Time
x1 ⋈ x1	1.00	1.00	1.04	1.25	1.25	1.00	1.00	1.00	1.08	1.81	1.96	1.18
x100 ⋈ x10M	2.75	33.11	11.37	1.34	1.18	1.22	1.00	1.13	1.03	1.93	1.08	1.15
x1K ⋈ x1M	1.17	32.85	12.15	1.08	4.82	2.52	1.00	1.05	1.05	1.63	1.25	1.16
x10K ⋈ x100K	1.04	32.83	11.03	2.02	32.81	11.29	1.00	1.02	1.10	1.57	1.09	1.08

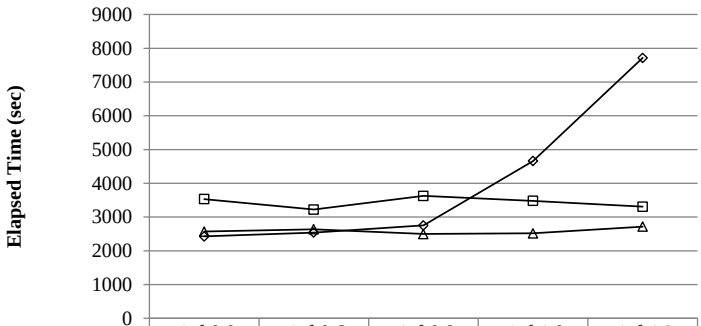
Table 2: Data Size - # of Records (GB)

		x1 ⋈ x1	x100 ⋈ x10M	x1K ⋈ x1M	x10K ⋈ x100K
M-IN	ALL	200 (42)	200 (42)	200 (42)	200 (42)
M-OUT	HASH	200 (43)	200 (43)	200 (43)	200 (43)
	RANGE	200 (44)	200 (44)	200 (44)	200 (44)
	RANDOM	1200 (263)	1200 (263)	1200 (263)	1200 (263)
	MDRP	200 (44)	200 (44)	238 (52)	261 (57)
R-OUT	ALL	100 (41)	1090 (449)	1098 (452)	1100 (453)

The Out column is most distinguishable than the other columns. We can see that HASH's poor performance is a result of the output skew. The maximum number of output tuples is greater over 30 times than the optimal number of output tuples. The RANGE algorithm detects and handles the output skew in the x100 $\bowtie$ x10M case, but data skew in the other cases are not detected. However, the RANDOM and MDRP algorithms prevent load imbalances in all cases.

In Table 1, it should be mentioned that the RANDOM algorithm is better than MDRP in terms of load balancing. All ratios are close 1 which is the optimal ratio. However, as shown in 9, the processing time of MDRP is better than that of RANDOM. This is because the RANDOM algorithm has to process more input tuples than MDRP. In this experiment, each relation contains 100M tuples, and the RANDOM algorithm duplicates a tuple $\sqrt{k} = \sqrt{36} = 6$ times. Therefore, the number of input tuples for reducers is 1.2 billion (600M for each relation). However, MDRP only produced 261M input tuples in total, which is about $1/4$ times, compared to the RANDOM algorithm.

Table 2 shows details of data size. M-IN represent the number of records (or bytes) of input data. M-OUT means the total number of output tuples from all mappers. The output tuples are fed to reducers. Finally, R-OUT records are stored into the distributed file system. With these results, we can see that the RANDOM algorithm generates many M-OUT tuples compared to the others.



	zipf-0.0	zipf-0.6	zipf-0.8	zipf-1.0	zipf-1.2
HASH	2428	2541	2754	4662	7715
RANDOM	3534	3224	3626	3480	3308
MDR	2570	2640	2498	2518	2717

Figure 10: Performance on Zipf's distribution

3.4.2 Zipf's Distribution

In this experiment, we create a pair of data sets (one for each join input). For one data set, the join attribute values are drawn uniformly at random from the range 1 - 1M. For other data set, we set the skew factor β from 0 (uniform) to 1.2. The skew factor is set to 1.0 in common, but we conduct more experiments in order to see a clear trend in results. Regardless of the skew factor, the number of output tuples is about 10 billion which occupies 4TB of disk space. As the number of tuples in a relation is 100M, each join attribute value has 100 tuples in a relation when the skew factor is 0.

In Figure 10, we change the skew factor of a data set, while the other data set has the same skew factor zipf-0.0. As predicted, we obtained a similar result with the scalar skew experiment. The HASH algorithm is vulnerable to the presence of data skew. The performance sharply degrades when the skew factor is greater than 0.8. This kind of data is very common in real

world such as word frequency, citation of papers, Web hits and so on [26]. The need for skew handling algorithms is reconfirmed. On the other hand, RANDOM and MDRP show consistent performance regardless of the skew factor. In the elapsed time, the MDRP algorithm outperforms the other algorithms. We do not present results of the RANGE algorithm because the results are similar with that of the MDRP algorithm. Since we only change skew factors of a data set, this result is reasonable.

3.4.3 Non-Equijoin Experiments

We now compare our algorithm with the state-of-the-art theta-join algorithm. In [8], the Cloud data set is used for evaluation of theta-join queries. Especially, two types of theta-join queries are examined: input-size dominated joins and output-size dominated joins. We employed the same queries for the comparison purpose. The input-size dominated test query is as follow:

```
SELECT R.date, R.longitude, R.latitude, S.latitude
FROM Cloud AS R, Cloud AS S
WHERE R.date = S.date AND R.longitude = S.longitude
      AND ABS(S.latitude - T.latitude) <= 10
```

This join produces 390 million output tuples, a much smaller set than the total of 382 * 2 million input tuples. On the other hand, the output-size dominated test query is:

```
SELECT R.latitude, S.latitude
FROM Cloud-5-1 AS R, Cloud-5-2 AS S
```

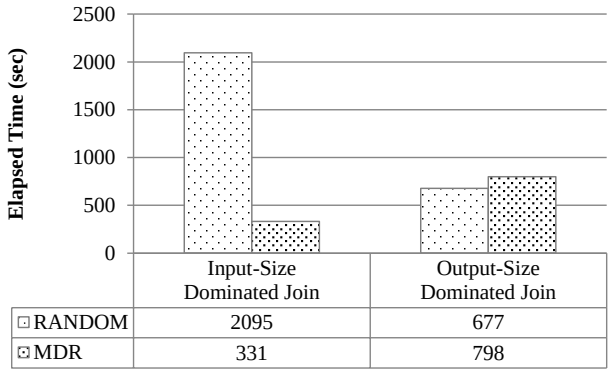


Figure 11: Performance on Non-Equijoin Queries

```
WHERE ABS(S.latitude - T.latitude) <= 2
```

For the output dominated query, we take two independent random samples of 5 million records each from Cloud (Cloud-5-1 and Cloud-5-2). The result contains 20 billion output tuples, a much larger set than the total 10 million input tuples. Again, we adopted all data sets and test queries from [8].

Figure 11 shows results of two test queries. We only report results of RANDOM and MDRP algorithms because HASH and RANGE do not deal with non-equijoin queries. In the input-size dominated query, it is clear that the MDRP algorithm outperforms the RANDOM algorithm. Since two algorithms produce the same join results, the difference in elapsed time can be seen the result of input duplications. If the input size grows, the performance gap will be larger and larger. However, in the output dominated query, RANDOM shows the better performance than ours. This is a reasonable result due to the large data size. In practice, the RANDOM algorithm actually provides near optimal solution in terms of load balancing. Our load balancing algorithm though also achieves the similar result.

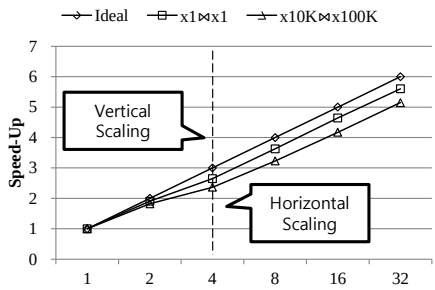


Figure 12: Speed-Up Experiments

In a single machine, the cross-product operation requires comparisons of n^2 pairs where n is the input size. In a shared-nothing system, the n^2 comparisons are distributed into multiple machines, but duplication of input tuples cannot be avoided. The MDRP algorithm filters non-candidate cells out so that we can reduce actual comparisons of n^2 pairs and makes smaller duplication compared to the RANDOM algorithm.

3.4.4 Scalability Experiments

Finally, we conduct speed-up and scale-up experiments. The speed-up means that as we add more machines by a certain factor, the time taken to compute a join operation should be decreased by the same factor. To measure the speed-up, we compute a join x1 $\bowtie$ x1 and x10K $\bowtie$ x100K that represent zero skew and skew cases respectively. We increase the number of cores (for mappers and reducers) from 1 to 32. From 1 to 4, we use the vertical scaling technique, i.e. we use a single machine that has multiple cores. From 4 to 32, we use the horizontal scaling technique, i.e. we use 8 machines each of which has 4 cores.

Table 3: Speed-Up Details

Core	1	2	4	8	16	32
x1 $\bowtie$ x1	8,837	4,716	2,816	1,429	707	364
x10K $\bowtie$ x100K	13,863	7,845	5,380	2,956	1,537	782

The results are shown in Figure 12 and Table 3. The 'ideal' line shows the linear speed-up, and we can see that our algorithm also follows the linear speed-up. This trend does not depend on the existence of data skew. However, it is notable that there is a difference between results of vertical and horizontal scaling techniques. In vertical scaling, the performance does not follow the linear speed-up. This indicates that there was an intervention between cores in a single node. In contrast, the performance in horizontal scaling shows the linear speed-up. This means that the input data is small so that the network overhead does not affect the performance.

The scale-up measures that as we add more machines, the size of a task that can be executed in a given time should be increased by the same factor. In our experiment, we add a machine that contains 4 cores, i.e. horizontal scaling. Since we deal with the join operation, we conduct two scale-up tests for input-size dominated joins and output-size dominated joins. For input-size dominated joins, we compute x1 $\bowtie$ x1 where the number of input tuples varies according to the number of machines (from 100M to 800M). In other words, for input-size dominated joins, we assume that each relation has the uniform distribution on the join attribute. On the other hand, in output-size dominated joins, we compute x10K $\bowtie$ xL where L varies from 100K to 800K. Therefore, the number of output tuples is greater than that of input tuples. The input size is fixed to 100M.

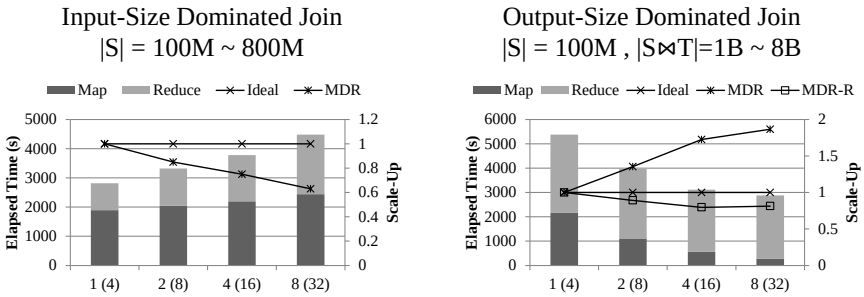


Figure 13: Scale-Up Experiments

Table 4: Scale-Up Details

# of Machines	1	2	4	8
Input-Size Dominated Join	2,816	3,320	3,378	4,479
Output-Size Dominated Join	5,380	3,972	3.119	2,883

The results are shown in Figure 13 and Table 4. In input-sized dominated join queries, the scale-up ratio is 0.6 when machines are increased from 1 to 8. In output-size dominated join queries, we found out an interesting aspect. The total elapsed time ‘MDRP’ shows better scale-up than the linear scale-up ‘ideal’. This is because we only change the number of output tuples, while the input size is fixed. More mappers can process the same input relations faster. We can see that the ’MDRP’ line increases linearly, which means that mappers also show the almost linear scale-up. The ‘MDRP-R’ line shows the average time for reduce tasks, and it almost follows the linear scale-up. We guess a reason of this is the low network cost.

3.5 Discussion

3.5.1 Sampling

To balance the workloads across reducers, we need to create equal sized cells. Since we determine sub-ranges via sampling, we need a sampling method that guarantees the size of a sub-range. Specifically, we approximate quantiles of given join attribute values which make each sub-range equal sized.

The Kolmogorov statistic [27] provides us a probabilistic guarantee for sampling quantiles. Suppose that we want to estimate the median value (at the 50% position in the sorted list of join attribute values). Let α be the proportion of tuples in a relation that have smaller join attribute values than the median value. Let β be the proportion of tuples in the sample that satisfy the same condition. The Kolmogorov's statistic tells us that $|\alpha - \beta| \leq d$ with probability $\geq p$ if the sample size is at least n. d is called the precision and p is the confidence. Given the values of p and d, n can be found using standard tables. Table 5 shows some representative cases, reproduced from [28]. For example, if we take 26,575 samples and select a value appeared at the 50% position, then with 99% confidence the value appears between 49% and 51% in the original relation. The Kolmogorov statistic works equally well for any

Table 5: Required sample size

d / p	0.90	0.95	0.98	0.99
0.10	149	185	234	266
0.05	596	740	937	1,063
0.01	14,900	18,500	23,425	26,575

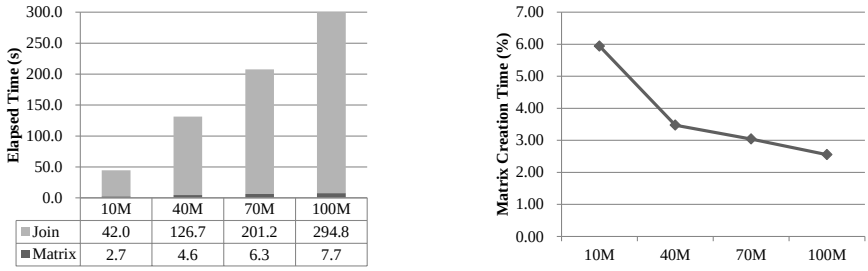


Figure 14: Elapsed Time for the Sampling

distribution and used in many literatures [29, 28]. In our experiment, we use $p = 0.99$, $d = 0.01$ and $n = 26,575$.

In our implementation, we assume that a relation is stored in HDFS with a random order on a join attribute. The HDFS partitions a relation into multiple splits according to a given block size (64MB). If q split needs to take n samples, each split takes n/q samples. Using HDFS's API, we call getSplits() method to obtain a list of splits. After that, we take n/q samples from the top of each split. Since we assume that the relation is sorted in a random order on the join attribute, this sampling method is sufficient for our purposes.

An important issue in the sampling process is the elapsed time for constructing the partitioning matrix. For this reason, we conduct an experiment to measure the matrix creation time. We compute x1 $\bowtie$ x1 and measure the ratio of the matrix creation time compared to the total elapsed time. The input data sets are not skewed because if there is a skewed value, the ratio of the matrix creation time would be smaller. The results are shown in Figure 14. We increase the size of input data set from 10M to 100M. Then, the ratio decreases from 6 to 3 percent. We think this ratio is affordable because gains

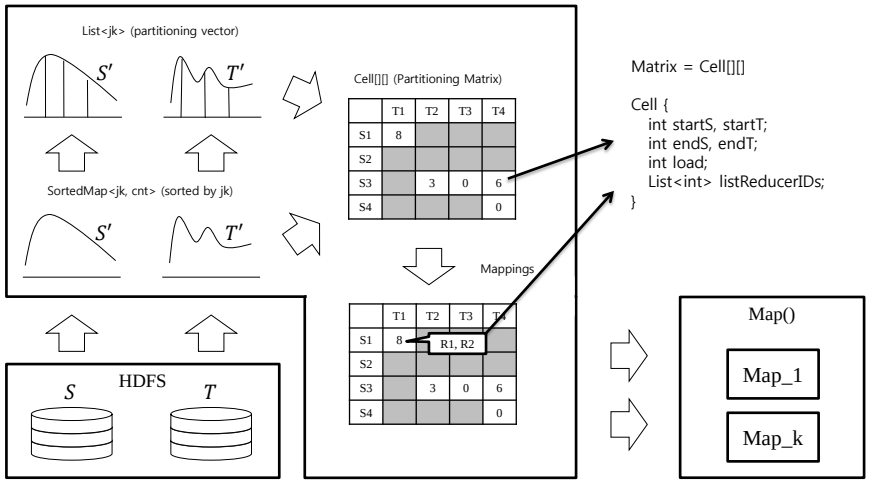
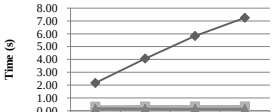


Figure 15: Processing Steps for the Sampling

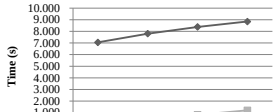
$|S'| = 26{,}575,\ k = 48$



	10M (64)	40M (272)	70M (457)	100M (626)
Sampling	2.17	4.07	5.84	7.25
Creation	0.32	0.34	0.34	0.34
Distribution	0.16	0.15	0.14	0.14

|S| (# of chunks, 64MB)

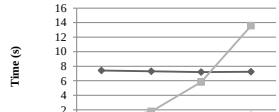
$|S| = 100\text{M},\ k = 48$



	10K	40K	70K	100K
Sampling	7.050	7.802	8.380	8.846
Creation	0.173	0.489	0.818	1.205
Distribution	0.120	0.130	0.131	0.165

# of samples

$|S| = 100\text{M},\ |S'| = 26{,}575$



	100	400	700	1000
Sampling	7.418	7.301	7.202	7.239
Creation	0.474	1.784	5.837	13.582
Distribution	0.172	0.385	0.752	1.281

# of cores (k)

Figure 16: Processing Times for Sampling Processes

from skewed input data is relatively bigger than the sampling cost.

To understand the sampling process, Figure 15 shows the processing steps of the partitioning matrix creation. We take samples from relations stored in the HDFS (sampling), and we create the partitioning matrix using the samples (creation). Finally, we distribute the partitioning matrix to all mappers (distribution). Therefore, we can analyze the sampling cost step by step.

Figure 16 shows processing times for sampling processes. For different input data size $|S|$, the sampling time is increased. For different number of sample size $|S'|$, the sampling time and the creation time is increased. However, the amount of increasing time is relatively small, so is seems to be possible to sample more records in order to create the partitioning matrix. Finally, the number of processing units k is important because the processing time is increased exponentially. When $k = 1000$, it require almost 15 seconds to create the partitioning matrix. If input size is small, this can be an overhead.

3.5.2 Memory-Awareness

Given k reducers, our approach initially create k^2 cells in the partitioning matrix. Since the MapReduce framework is supposed to handle very large data sets, the size of a cell is sometimes too big to fit in memory. We can avoid this situation by making the implementation 'memory-aware'. Suppose that we have a memory limit m, the maximum number of input tuples fit in the memory. Two input relations S and T use the memory $m/2$ respectively. Given the maximum size of a relation $max(|S|, |T|)$, we can simply select the number of sub-ranges for each relation, i.e $p = \lceil max(|S|, |T|)/(m/2) \rceil$. Then, the partitioning matrix contains p^2 cells which fit in the memory.

It is difficult to determine p without knowing the size of input relations $|S|$ and $|T|$. In Hadoop, we can effectively estimate the size with small samples. When we take samples as described in Section 3.5.1, we use the RecordReader class which is connected with data splits. The class provides the getProgress() method which informs the progress of reading the split.

Suppose that n/q sample tuples occupy γ percent of a split. Then, the number of input tuples in the split is $q/n \cdot \gamma$. Since we have q splits, the total number of input tuples in a relation is $q^2/n \cdot \gamma$.

3.5.3 Handling of Heavy Cells

We have defined a heavy cell and propose a technique to handle the heavy cell. We chop the heavy cell so that many computing nodes can process the heavy cell simultaneously. However, one can have a question about the handling of heavy cells. For example, we can divide a heavy cell into smaller cells in terms of input size. The smaller cells are involved to smaller ranges of join key attribute values. Hence, the number of output tuples in a cell can be decreased. An exceptional situation is that a join key attribute value overwhelms the other values. For instance, in a scalar skew data set, there are a number of input tuples having '1' in its join attribute whereas the other values follow the uniform distribution. In this case, a heavy cell containing '1' in its range cannot be divided by smaller sub-ranges. The divided cell will also be a heavy cell.

In Figure 17, we show the effectiveness of small cells. In this experiment, we have computed x10K $\bowtie$ x100K. The column '1' is the original setting, and '1/2' represent the size of sub-range compared to the original setting. We can see that the smaller cells does not affect to the elapsed time of the join operation. The reason is that '1' values cannot be divided as we have already discussed. Another interesting point is that M-OUT is decreased as the size of cell is decreased. Using the fragment-and-replicate technique, a cell can be replicated to several reducers. Therefore, small cells

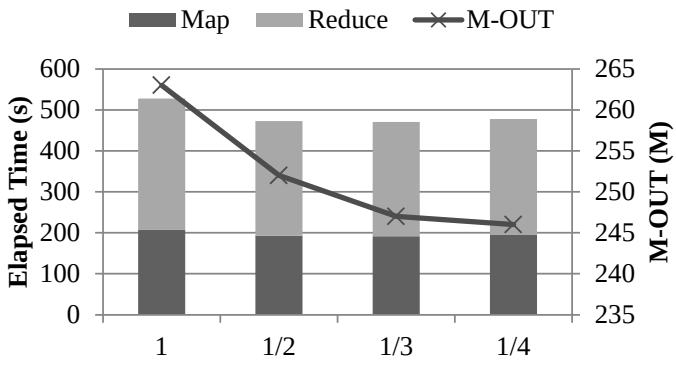


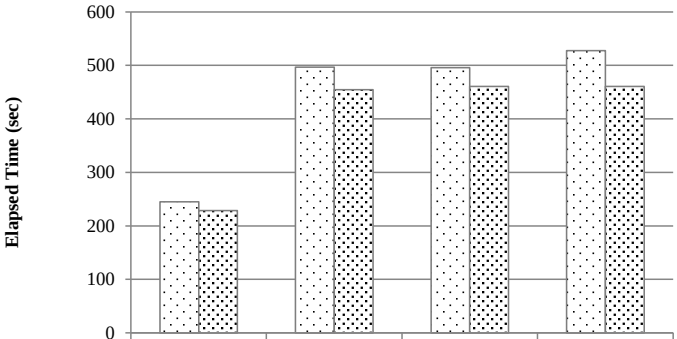
Figure 17: Dividing Heavy Cells with Small Sub-Ranges

reduce the number of tuples that have to be replicated.

3.5.4 Existing Histograms

In MDRP, the sampling process is required in order to create the partitioning matrix. However, if we maintain histograms of input data, we do not have to perform the sampling process. This reduces the processing time of overall join evaluation. Moreover, exploiting histograms is helpful for load balancing. A number of studies have proposed algorithms to create and maintain an equi-depth histogram. In the equi-depth histogram, the number of data records in a bucket is similar with each other. Therefore, the data buckets are similar with a cell in a partitioning matrix, and by assigning buckets to the reducers, we can expect the optimal balance of workloads.

Exploiting histograms provides fine-grained balancing of workloads. Without histogram, we only are able to use a range instead of join attribute value itself. This yields unnecessary data replication as we have shown in



	x1⋈x1	x100⋈x10M	x1K⋈x1M	x10K⋈x100K
MDR-CELL	245	497	496	528
MDR-KEY	229	455	461	461

Figure 18: Exploiting Histograms

the previous subsection. To measure benefits from exploiting histograms, we compare the elapsed times as shown in Figure 18. MDRP-CELL indicate the original algorithm without exploiting histograms. It is notable that in this experiment, MDRP-CELL does not contain sampling time because we want to measure the effectiveness of fine-grained balancing. MDRP-KEY shows the elapsed time when we exploit existing histogram. Compared to MDRP-CELL we can see that MDRP-KEY shows significant improvements. For example, in x10K $\bowtie$ x100K data sets, the improvement is $(1-(461/528))*100 = 12.69\%$.

The experimental results show that exploiting of histogram is helpful for join processing. However, building and maintaining a histogram is a difficult problem. In this case, spatial indexing structures can be used as a histogram. For example, the grid file system [30] provides an overview of underlying data. The grid file system create the grid directory that represents and maintains the dynamic correspondence between record space and data

buckets. The grid directory consists of two parts: a dynamic k-dimensional array which contains pointers to data buckets and k one-dimensional arrays which define a partition of a domain. Therefore, we can obtain the exact number of records in a bucket (or a cell) by counting the number of pointers.

Since the grid file maintains sub-ranges of attribute values, it seems to be seen similar with our approach. However, there are many different points: 1) a dimension of the grid file represents an attribute in a relation. In our approach, different dimensions correspond to different relations. 2) The main purpose of our approach is processing of parallel joins instead of creating of persistent indexes. Therefore, our partitioning matrix is created on-the-fly whenever a join operation is evaluated. 3) We introduce the notion of heavy cells and propose a technique to chop the heavy cells. Our heavy cell definition is based on the join selectivity (result size) which is a feature that only exists in the join operation. On the other hand, the spatial indexing structure create partitions using their sub-ranges. Different sub-ranges can have different width. Hence, the heavy cells can be viewed as a wide sub-range. In join operations, this difference can cause input imbalance across reducers.

3.6 Summary

Handling data skew is essential for efficient join algorithms using MapReduce. The range-based and randomized partitioning approaches have been widely used so far, but they also have some limitations. In this paper, we proposed a new approach that outperforms traditional approaches. Our ap-

proach is better than the range-based approach when join product skew exists in underlying data. This is because we consider samples from all relations to be joined, while the range-based approach only considers samples from the most skewed relation. In addition, our approach outperforms the randomized approach when the size of input relation becomes large. The reason for this is that the randomized approach creates multiple copies of input data regardless of given join conditions. Through extensive experiments over synthetic and real world data, we demonstrated the effectiveness of our proposed approach.

Chapter IV

Extensions

In this chapter, we apply the MDRP technique to join operations involved in multiple relations, i.e. multi-way joins. Joining multiple relations are common in many data analysis tasks. In Section 4.1, we first introduce some important applications that require multi-way joins. The applications contain the graph pattern matching and the matrix chain multiplication. With these two examples, we show that a multi-way join query can be decomposed into two sub-join queries: single-key based join (SK-Join) and multiple-key based join (MK-Join). In Section 4.2, we explain how the proposed technique can be applied to each join type.

Efficient processing of multi-way joins using MapReduce contains several issues including handling of data skew. In this chapter, we address another issue for multi-way joins, i.e. handling of complex queries (combinations of SK-Join and MK-Join). In Section 4.3, We review iteration-based and replication-based algorithms to handle complex queries, and we then propose a join-key selection algorithm combining iteration-based and replication-based algorithms in Section 4.4. After that, we take Section 4.5 in order to demonstrate the effectiveness of our skew handling technique for multi-way joins.

4.1 Joining Multiple Relations in a MapReduce Job

In this section, we introduce two representative applications that require joins of multiple relations: graph pattern matching and matrix multiplications. These applications are related to graph analysis tasks. Actually, graph analysis includes a number of multi-way joins because of its underlying triple (subject, predicate, object) data model. It requires many self-joins when we try to analyze relationships between entities. Moreover, graphs usually have significant skew in terms of the degree distribution, e.g. scale-free networks [31]. With these practical examples, we identify important sub-types of multi-way joins: SK-Join and MK-Join.

4.1.1 Example: SPARQL Basic Graph Pattern

Processing of SPARQL Basic Graph Pattern (BGP) [32] is a good example of the graph pattern matching problem. SPARQL is a query language for RDF data [33] which is W3C's standard graph representation format. A SPARQL query contains a BGP which is a set of triple patterns. For example, Figure 19 contains a BGP that consists of 5 triple patterns (TP) which have a shared variable x. As shown in the example, each triple pattern has to be joined with each other, thereby efficient processing of multi-way join is very essential. Especially, the example query is an exact SK-Join query which we will see in Section 4.1.3.

Basically, the example query requires four two-way joins, as expressed in Figure 20(a). However, the MapReduce framework can evaluate the query

SELECT ?x ?y1 ?y2 ?y3
WHERE {

BGP

?x rdf:type ub:Professor.	TP#1
?x ub:worksFor <Department0>.	TP#2
?x ub:name ?y1.	TP#3
?x ub:emailAddress ?y2.	TP#4
?x ub:telephone ?y3	TP#5

}

S	P	O
Student_0	type	Student
Student_0	name	“Student_0”
Student_0	memberOf	Department_0
Student_0	takesCourse	Course_0
Student_0	takesCourse	Course_1
Student_1	type	Student
Student_1	name	“Student_1”
Student_1	memberOf	Department_0
Student_1	takesCourse	Course_0
Student_2	type	Student
Student_2	name	“Student_2”

Figure 19: Joins in graph pattern matching queries

SELECT ?x ?y1 ?y2 ?y3
WHERE {
1 ?x rdf:type ub:Professor.
2 ?x ub:worksFor <Department0>.
3 ?x ub:name ?y1.
4 ?x ub:emailAddress ?y2.
5 ?x ub:telephone ?y3
}

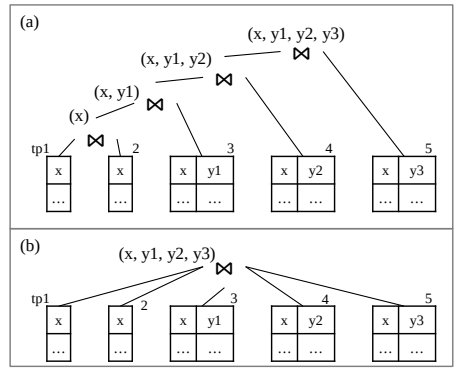


Figure 20: Two-way joins vs. multi-way joins

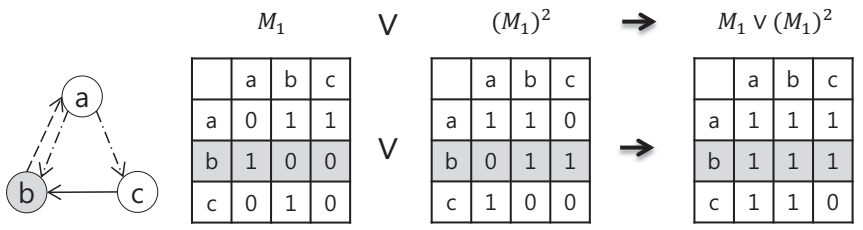


Figure 21: An example of powers of a matrix

all at once as in the five-way join expressed in Figure 20(b). In MapReduce, a multi-way join can be viewed as a MapReduce job that contains multiple joins in it. As RDF has a simple triple data model, it is not unusual that a SPARQL query includes several triple patterns having the same variable as shown in Figure 19. Hence, dealing with SK-Join in MapReduce is very common in many data analysis tasks.

4.1.2 Example: Matrix Chain Multiplication

Matrix chain multiplication is another example of multi-way joins. Actually, matrix operations are also substantially related to many graph algorithms. In Figure 21, we consider Warshall's transitive closure algorithm [34]. As shown in the example, we have a graph and a zero–one adjacency matrix that represents the graph. Then, the union of powers of the adjacency matrix produces the transitive closure. The computation of PageRank [35] can be another example. It requires iterative matrix-vector multiplications, which can be seen as the matrix chain multiplication problem.

Actually, the semantics of matrix multiplication can be implemented by the join operation in database systems [36]. Figure 22 shows an example.

We have two n by n matrices M_1 and M_2. The (i, j)-th element of $M_1 \times M_2$ is $\sum_{j=1}^{n} m_{1_{i,j}} \times m_{2_{j,i}}$. Now suppose that we have two relations R_1 and R_2 that have the same attributes $\{row, col, val\}$. A record in a relation corresponds to a non-zero element in a matrix. We then are able to express a binary multiplication in terms of SQL.

```
SELECT R1.row, R2.col, sum(R1.val*R2.val)
FROM R1, R2
WHERE R1.col=R2.row
GROUP BY R1.row, R2.col
```

The result of this query represents a matrix whose $(R_1.row, R_2.col)$-th element is a sum of $(R_1.val * R_2.val)$. Although representation schemes can be different, the results of a multiplication are equivalent. We denote this relational expression of matrix multiplication by $R_1 \bowtie\!\!\!\!\!\!\lhd_* R_2$.

The multi-way join operation allows us to combine multiple, separate two-way joins, meaning that we can compute $(R_1 \bowtie\!\!\!\!\!\!\lhd_* R_2 \bowtie\!\!\!\!\!\!\lhd_* R_3)$ at once. Since join conditions in $R_1 \bowtie\!\!\!\!\!\!\lhd_* R_2$ and $R_2 \bowtie\!\!\!\!\!\!\lhd_* R3$ are different, we can see that this is a 'pure' MK-Join as we will see in Section 4.1.3.

We focus on a single multiplication that evaluates $M_1 \times M_2 \times M_3$ in parallel. Since matrix multiplication is associative, this approach helps to improve the overall performance of an algorithm. However, it is difficult to optimize the multi-way join operation in MapReduce because of the I/O and communication overhead despite the research done in this field [13], [14]. Therefore, we will examine existing multi-way join algorithms in order to improve the performance of matrix chain multiplication.

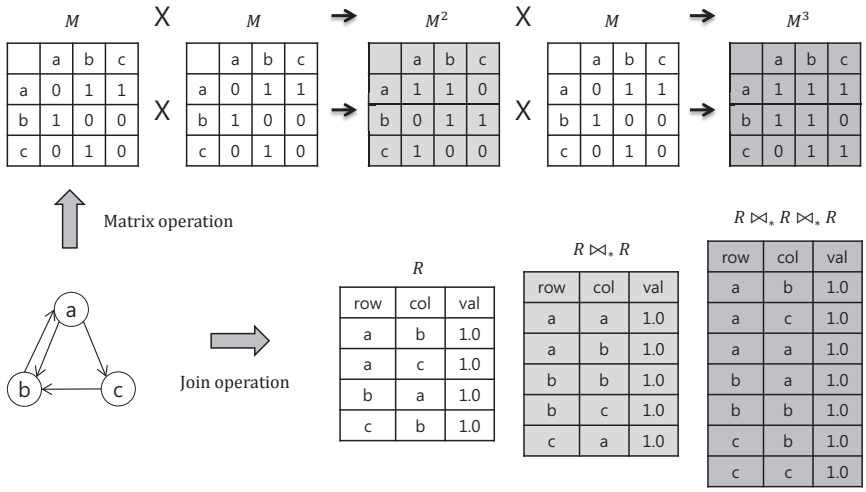


Figure 22: Join operations in matrix multiplications

4.1.3 Single-Key Join and Multiple-Key Join Queries

As shown in previous sections, a multi-way join queries can be decomposed into two types of sub queries: SK-Join and MK-Join. We first define SK-Join and MK-Join.

Definition 2. (SK-Join) *Let R be a set of relations. A single-key join is a join operation between a subset of R in which all relations have a single join attribute, jk.*

In other words, let us suppose that we have a set of join-keys JK for a multi-way join operation. As represented in its name, SK-Join query has only a single join-key jk in JK. Therefore, all relations are related to each other via a single join-key attribute jk, thereby can be joined in a MapRe-

duce job. For example:

$$R(A,B) \bowtie S(A,C) \bowtie T(A,D) \bowtie U(A,E)$$

is a SK-Join because they can be joined by a join key attribute A.

It should be noted that the definition of the SK-Join is a special case of the traditional star-joins performed over the star-schema. For example,

$$R(A,B,C,D) \bowtie S(B,I) \bowtie T(C,J) \bowtie U(D,K)$$

is a traditional star-join query. However, this special case is also involved to multiple relations, and we want to distinguish this special case from chain-joins which are involved in multiple join attributes. Hence, in this paper, we denote the special case by SK-Join.

Definition 3. (MK-Join) *Let $|jk|$ be the number of relations joined by the given join-key attribute jk. A multiple-key join query is a join operation in which $|jk|$ is at most 2 for all $jk \in JK$.*

For example, a pure chain-join operation implies several 2-way joins as follows:

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D) \bowtie U(D,E)$$

is a MK-Join query. A set of join keys is $JK = \{B,C,D\}$, and $|B| = |C| = |D| = 2$. This type of query requires iterative MapReduce jobs or replication of input relations.

It should be noted that both SK-Join and MK-Join can exist together

in a query. Thus we should consider combinations of query types as well as SK-Join and MK-Join queries separately. We first deal with two joins in Section 4.2, and combinations of joins are discussed in Section 4.3.

4.2 Skew Handling for Multi-Way Joins

We now discuss how the proposed skew handling technique, MDRP, can be applied to multi-way join queries. As we have seen, multi-way join queries can be decomposed into two sub-classes. Therefore, we deal with each join types in different subsections.

4.2.1 Skew Handling for SK-Join Queries

We first consider a SK-Join query among three relations R, S and T. Suppose that they have the same join key jk. Then we can create a partitioning cube instead of the partitioning matrix. In other words, the partitioning matrix is a two-dimensional space between two relations, but the partitioning cube is a three-dimensional space among three relations.

Figure 23 shows an example of the partitioning cube. Each relation is projected to a dimension of the partitioning cube. Each cell indicates a sub-range considering three relations that is participated in the given join operation. Then, the other processing steps are the same with the processing steps of two-way joins. We first find heavy cells. If exists, we chop the heavy cells into non-heavy cells and create a mapping between cells and reducers. In join phase, we use the fragment-replicate technique so that we can obtain correct join results.

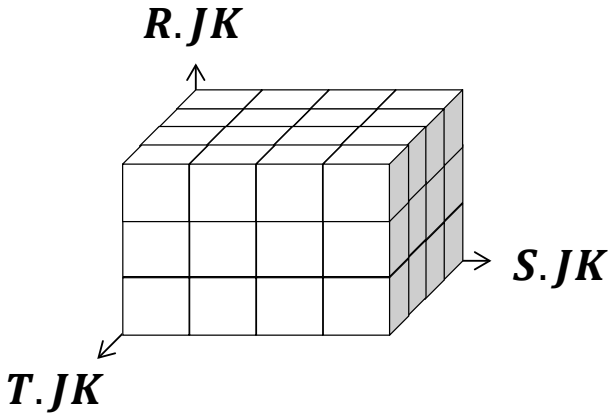


Figure 23: Skew handling in SK-Join queries

In the randomized approach, it is notable that multi-way SK-Join requires more input duplication than two-way join queries. To produce correct join results, we fragment a relation while we replication the other two relations. Therefore, we have to duplicate two relations. Suppose that we have k reducers and m relations. Then, we have to make $k^{(m-1)/m}$ duplications for replicate relations. Therefore, the randomized approach is not a good choice for multi-way join queries. We will discuss more details about this issue in Section 4.5.1.

4.2.2 Skew Handling for MK-Join Queires

We now consider a MK-Join query among four relations R, S, T and U. In this case, we have join keys which are only related to two input relations. It is worthwhile to mention that it is difficult to compute the entire joins in a MapReduce job. However, it is at least possible that we can make all input relations to participate in a MapReduce job. In other words, computing

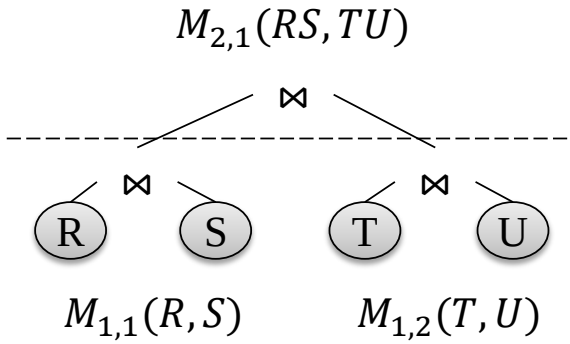


Figure 24: Skew handling in MK-Join queries

$(R \bowtie S \bowtie T \bowtie U)$ in a job is difficult, while computing $(R \bowtie S)$ and $(T \bowtie U)$ in a job is relatively simple. We will compare these approaches in the next section, and in this section, we use the second approach.

To compute $(R \bowtie S)$ and $(T \bowtie U)$ in the same MapReduce job, we can create two partitioning matrices (or cube) using samples from all input relations. Figure 24 shows an example of processing steps for the given chain-join. In the first MapReduce job, we create two partitioning matrix $M_{1,1}(R,S)$ and $M_{1,2}(T,U)$ respectively, where $M_{i,j}(R,S)$ represents j-th matrix for $R \bowtie S$ in i-the MapReduce job. Note that, input tuples from different input relations can be distinguished from each other because the input tuples have their own tag attached from the Map function. Therefore, a Reduce function can compute several join operations simultaneously. But, we have to consider the size of input relation for each reducer in order to avoid memory overflows.

After the first MapReduce job is finished, we have two joined realtions $R \bowtie S$ and $T \bowtie U$. Therefore, in the second MapReduce job, we can create

$M_{2,1}(RS, TU)$ as in typical two-way joins in order to evaluate the final join operation.

4.3 Combinations of SK-Join and MK-Join

In this section, we deal with complex queries defined in subsection 4.3.1. Simply, complex queries are combinations of MK-Join and MK-Join.

To process a complex query, we need an efficient algorithm because a MK-Join query cannot be processed using a single MapReduce. Algorithms for complex queries can be categorized into two classes: iteration-based and replication-based algorithms. We take subsection 4.3.2 and 4.3.3 in order to explain the algorithms.

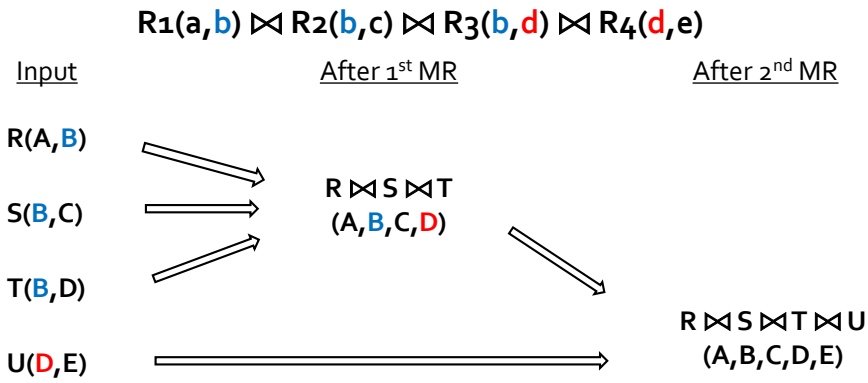
Although replication-based algorithm works well in small data sets, many real-world data contains huge amount of data that should not be duplicated. In Section 4.3.4, we compare two approaches in order to understand the trade-off between iteration and replication costs.

4.3.1 Complex Queries

Simply, a complex query is a combination of SK-Join queries and MK-Join queries. Therefore, the number of distinct join-keys should be greater than or equal to 2. Let us consider the following example query:

$$R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(b,d) \bowtie R_4(d,e)$$

This query contains four input relations and two join-key attributes $\{b,d\}$. Therefore, the query be seen as a complex query.



	Cascade of Multi-way joins
A	$A_1 = \{b\},\ A_2 = \{d\}$
P	$P_1(b) = \{R_1, R_2, R_3\},\ P_2(d) = \{J(1,2,3), R_4\}$

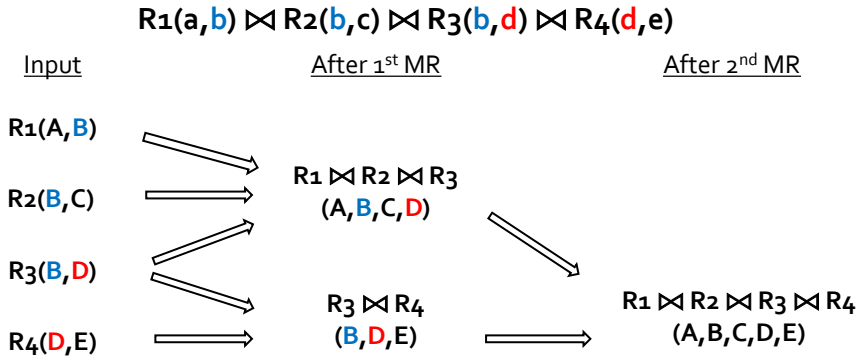
Figure 25: Cascade of multi-way joins

Note that, typical two-way join algorithms cannot process the complex query within a MapReduce job. This is because the query involves in two or more join-key attributes. Therefore, we should employ an algorithm for handling multi-way joins.

4.3.2 Iteration-Based Algorithms

A basic algorithm for processing complex queries is to evaluate 2-way joins step-by-step. We refer this algorithm to serial 2-way joins or cascade of 2-way joins. This algorithm is applicable to any complex queries. However, it does not take advantages of MapReduce's key-equality based data flows.

An alternative algorithm of the serial 2-way joins is the serial (cascade of) m-way joins. This iteration-based algorithm process relations that have the same join key attribute in a MapReduce job. In the above example, can-



	Cascade of Multi-way joins
A	$A_1 = \{b, d\},\ A_2 = \{b\}$
P	$P_1(b) = \{R_1, R_2, R_3\},\ P_1(d) = \{R_3, R_4\},\ P_2(b) = \{J(1,2,3), J(3,4)\}$

Figure 26: Cascade of parallel multi-way joins

didate join keys are b and d. Hence, we can first select b as a join key and create a partial result $R_1 \ltimes R_2 \ltimes R_3$. In the next MapReduce job, we select d as a join key and create the final result. Compared with the cascade of two-way joins, it differs in the perspective of joining multiple relations at once.

We have another alternative iteration-based algorithm so called the cascade of parallel multi-way joins. Figure 26 shows an example of the cascade of parallel multi-way joins. For each MapReduce job, we select all join keys. In the first MapReduce job, we select b and d and process $R_1 \ltimes R_2 \ltimes R_3$ and $R_3 \ltimes R_4$ simultaneously. In the second MapReduce job, we can produce the final result with a simple two-way join algorithm.

Note that the cascade of parallel multi-way join algorithm also incurs the replication of input relations (e.g. R_3 in the example). However, the amount of replication is relatively small and both replica contribute to pro-

duces the final results. On the other hand, in the replication-based algorithm, some replica do not used at all.

4.3.3 Replication-Based Algorithms

To explain the replication-based multi-way join algorithm [13], let us consider a join:

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$$

The Map function processes send each tuple of R and T to many different Reduce functions, although each tuple of S is sent to only one Reduce function. This is because S contains both join-keys whereas R and T has a partial single join-key. The duplication of data increases the communication cost above the theoretical minimum, but in compensation, it does not have to communicate the result of the first join. As we can see, the replication-based multi-way join algorithm can therefore be preferable if the typical tuple of one relation joins with many tuples of another relations, as would be the case, for example if we join copies of the matrix of the Web. In that case, the size of intermediate results is too large to temporarily store in to the shared storage.

As shown in Figure 27, suppose that we have $k = m^2$ reducers for $m = 4$. Values of B and C (join-keys) will each be hashed to m buckets, and each Reduce function will be associated with a pair of buckets, one for B and one for C. That is, we choose to make B and C part of the map-key, and we give them equal shares.

Let h be a hash function with range $1,2,...,m$, and associate each re-

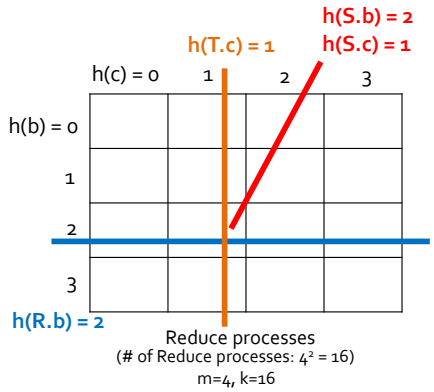


Figure 27: Replication-based multi-way join algorithm

ducer with a pair (i, j), where integers i and j are each between at the same range. Each tuple $S(b,c)$ is sent to the Reduce function numbered $(h(b),h(c))$. Each tuple $R(a,b)$ is sent to all reducers numbered $(h(b),x)$, for all x. Each tuple $T(c,d)$ is sent to all reducers numbered $(y,h(c))$ for any y. Thus, each reducer (i, j) gets $1/m^2$-th of S, and $1/m$-th of R and T.

Each reducer computes the join of the tuples it receives, It is easy to observe that if there are three tuples $R(a,b)$, $S(b,c)$, and $T(c,d)$ that join, then they will all be sent to the Reduce function numbered $(h(b),h(c))$. Thus, the algorithm computes the join correctly. However, we can see that input tuples duplicated to other reducers are discarded. This unnecessary communicate can be a burden to the system.

4.3.4 Iteration-Based vs. Replication-Based

We now compare I/O cost and communication cost of iteration-based and replication-based algorithms. We assume that a MapReduce job con-

sists of two operations related to disk I/O and a network-related operation. In other words, (1) an operation that reads input tuples from relations stored in a distributed file system; (2) an operation that sends input tuples to appropriate reducers, i.e. a network part of the shuffle phase (3) an operation that write the final join results from each reducer to the distribution file system. Actually, this assumption is not exactly matched to the processing steps of the Hadoop MapReduce framework. However, in the conceptual level, we believe that this abstraction is still useful.

We now define some terminologies in order to conduct the cost analysis. Suppose that we want to compute an equi-join operation among relations $R = \{R_1, R_2, ..., R_n\}$. For simplicity reasons, we denote $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ by $J(1,2,...n)$. In i-th MapReduce job, relations in a partition P_i are joined by a set of join key attributes A_i. In other words, $A_1 = \{a,b\}$ represents that join attributes for the first MapReduce job are a and b. On the other hand, P_i is a map that contains sets of relations with an associated join key attribute. In addition, $P_i(a)$ is a function from a join key attribute a to a set of joined relations by the join attribute.

Example 5 *(Partition P_i)* Let us consider an example partition $P_1 = [(a \rightarrow \{R_1, R_2\}), (b \rightarrow \{R_2, R_3\})]$. This indicates that the first MapReduce job contains two join key attributes a and b. And participating relations for the join key a in the first MapReduce job is denoted by $P_1(a) = \{R_1, R_2\}$. Similarly, $P_1(b) = \{R_2, R_3\}$. □

We now consider an example join $R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(b,d) \bowtie R_4(d,e)$. Then, I/O cost of four different multi-way join algorithms can be

	Cascade of 2-way joins	Cascade of multi-way joins	Cascade of parallel multi-way joins	One-shot multi-way join
P	$P_1(b) = \{R_1, R_2\}$ $P_2(b) = \{J(1,2), R_3\}$ $P_3(d) =$ $\{J(1,2,3), R_4\}$	$P_1(b) = \{R_1, R_2, R_3\}$ $P_2(d) =$ $\{J(1,2,3), R_4\}$	$P_1(b) = \{R_1, R_2, R_3\}$ $P_1(d) = \{R_3, R_4\}$ $P_2(b) =$ $\{J(1,2,3), J(3,4)\}$	$P_1(b,d) =$ $\{R_1, R_2, R_3, R_4\}$
Read	$\sum_{i=1}^{\|A\|}\left(\left(\sum_{R_n \in P_i(a), a \in A_i} \|R_n\|\right)\right)$			
Write	$\sum_{i=1}^{\|A\|}\left(\left(\sum_{a \in A_i} \|JOIN(P_i(a))\|\right)\right)$			

Figure 28: I/O cost analysis of multi-way join algorithms

estimated as shown in Figure 28. The cascade of 2-way joins consists of three phases of MapReduce jobs. In the first job, $P_1(b)$ is $\{R_1, R_2\}$. The result $J(1,2)$ is joined with R_3 in the second MapReduce job. Similarly, $P_3(d)$ consists of $J(1,2,3)$ and R_4. Likewise, all iteration-based algorithms require multiple MapReduce jobs. On the other hands, the replication-based one-shot multi-way join requires a single MapReduce job, i.e. $P_1(b,d) = \{R_1, R_2, R_3, R_4\}$. We can summarize that every i-th iteration reads all relations that are participated into the iteration, and every join results of each iteration should be materialized into the disk. The input and output costs then be formulated as follows:

$$Input : \Sigma_{i=1}^{|A|} \left(\Sigma_{R_n \in P_i(a), a \in A_i} |R_n|\right)$$

$$Output : \Sigma_{i=1}^{|A|} \left(\Sigma_{a \in A_i} |J(P_i(a))|\right)$$

We can see that the I/O cost becomes heavier as the number of iteration becomes large. In case of replication-based algorithm, it only read all input relations once, and then it also writes the final result only once. Therefore,

I/O cost of iteration-based algorithms is bigger than that of the replication-based algorithm.

Next, let us consider the communication costs. The communication cost should be distinguished by processing steps of a MapReduce job. Actually, read and write operations also need network overhead because input relations are stored in a networked storage. In addition, in the shuffle phase of a MapReduce job, we have to consider intermediate key-value pairs from mappers to reducers.

	Cascade of 2-way joins	Cascade of multi-way joins	Cascade of parallel multi-way joins	One-shot multi-way join
Read Write	$\sum_{i=1}^{\lvert A\rvert}\left(\left(\sum_{R_n\in P_i(a),a\in A_i}\lvert R_n\rvert\right)+\left(\sum_{a\in A_i}\lvert JOIN(P_i(a))\rvert\right)\right)$			
Shuffle	$\sum_{i=1}^{\lvert A\rvert}\left(\left(\sum_{R_n\in P_i}dup(R_n)\lvert R_n\rvert\right)\right)$			
P	$P_1(b)=\{R_1,R_2\}$ $P_2(b)=\{J(1,2),R_3\}$ $P_3(d)=$ $\{J(1,2,3),R_4\}$	$P_1(b)=\{R_1,R_2,R_3\}$ $P_2(d)=$ $\{J(1,2,3),R_4\}$	$P_1(b)=\{R_1,R_2,R_3\}$ $P_1(d)=\{R_3,R_4\}$ $P_2(b)=$ $\{J(1,2,3),J(3,4)\}$	$P_1(b,d)=$ $\{R_1,R_2,R_3,R_4\}$
Shuffle	1: $\lvert R_1\rvert+\lvert R_2\rvert$ 2: $\lvert J(1,2)\rvert+\lvert R_3\rvert$ 3: $\lvert J(1,2,3)\rvert+\lvert R_4\rvert$	1: $\lvert R_1\rvert+\lvert R_2\rvert+\lvert R_3\rvert$ 2: $\lvert J(1,2,3)\rvert+\lvert R_4\rvert$	1: $\lvert R_1\rvert+\lvert R_2\rvert+$ $2\lvert R_3\rvert+\lvert R_4\rvert$ 2: $\lvert J(1,2,3)\rvert+$ $\lvert J(3,4)\rvert$	1: $\lvert\mathrm{h(d)}\rvert(\lvert R_1\rvert+$ $\lvert R_2\rvert)+\lvert R_3\rvert+$ $\lvert h(b)\rvert\lvert R_4\rvert$

Figure 29: Communication cost analysis

Figure 29 shows the result of communication cost analysis. In the first MapReduce job of cascade of 2-way joins, the shuffle phase have to transfer $|R_1| + |R_2|$ input tuples from mappers to reducers because $P_1 = \{R_1, R_2\}$. Similarly, all iteration-based algorithms transfer input tuples to an appropriate reducer. However, replication-based algorithm makes copies of input tuples. For R_1 and R_2, we have to make $|h(d)|$ copies. We do not have to make copy of R_3, but R_4 have to be copied $|h(b)|$ times. Therefore, the communication cost incurred by the shuffle phase can be written as follow:

$$\Sigma_{i=1}^{|A|} \left(\Sigma_{R_n \in P_i} dup(R_n) \cdot |R_n| \right)$$

It should be noted that $dup(R_n) = 1$ for iteration-based algorithms, while $dup(R_n) = h(a)$ for the replication-based algorithm. In [13], the duplication factor $dup(R_n)$ is called *share*. The communication cost of the replication-based algorithm is clearly bigger than that of iteration-based algorithms.

4.4 Join-Key Selection Algorithms for Complex Queries

In our previous work [14], we have proposed two join key selection algorithms for processing multi-way join queries. With an example of SPARQL Basic Graph Pattern (BGP) in Section 4.1.1, we re-emphasize the importance of join-key selection algorithms.

It is worthwhile to note that we can determine join-keys for a complex query before the actual join begins. Using schema of data and the given

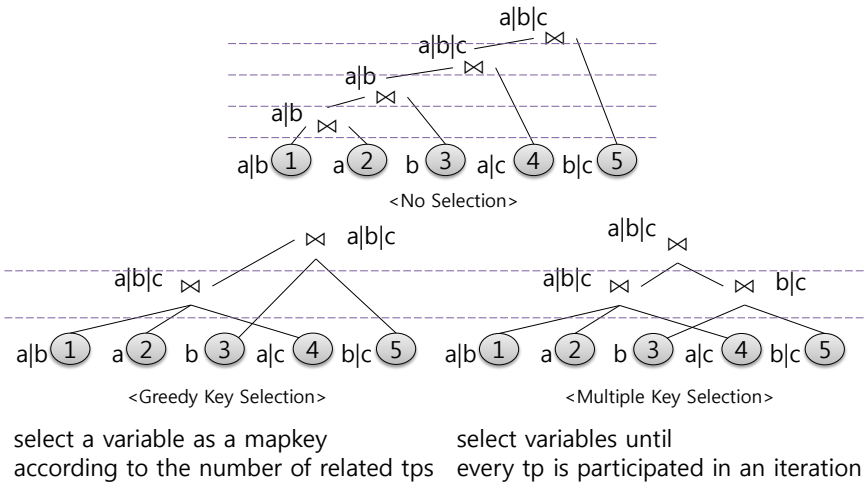


Figure 30: Join key selection strategies

query, we can extract candidate join-keys in the join operation and create a query execution plan. In a basic graph pattern in SPARQL, we can easily determine a join-key by selecting a shared variable because a join-key should be a shared variable. However, it is difficult to select a join-key when a BGP has two or more shared variables. In these complex queries, a join-key cannot cover all triple patterns. This implies that iterations of MapReduce jobs will occur. To minimize the number of MapReduce iterations, we have employed two heuristic join-key selection strategies: Greedy-Selection and Multiple-Selection. In this section, we describe these join-key selection algorithms.

4.4.1 Greedy Key Selection

Suppose that we have several join-key candidates which are variables that appear in two or more triple patterns. With the greedy selection strategy,

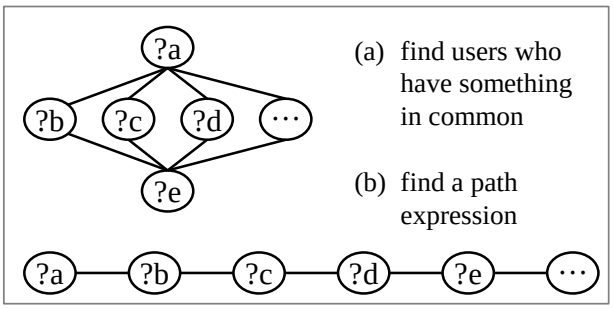


Figure 31: Typical examples of complex queries

we select a variable according to the number of related triple patterns. In other words, we sort the candidate join-keys according to the number of related triple patterns in an decreasing order. The greedy selection strategy benefits from the multi-way join technique. The strategy accelerates eager evaluation of multiple triple patterns, which is a specialty of the multi-way join technique (handling star-joins first). At least, it will not lead to a loss when reducing the number of MapReduce iterations. Figure 31(a) shows a case in which the greedy selection is particularly useful. Variables b, c, and d are join-key candidates because they are related to two triple patterns. If we select one of them, we can join only three variables: (a, b, e), (a, c, e), and so on. On the other hand, variables a and e are related to three or more triple patterns; hence, we can merge many variables all at once. The result will be the same, but the number of MapReduce iterations will be different.

4.4.2 Multiple Key Selection

MapReduce is a parallel processing framework implying that we can select multiple join-keys in a MapReduce job. It has an effect on resource

utilization. Let us consider the query shown in Figure 31(b). The query appears as a path expression; this is one of the most commonly used query patterns. Particularly, a RDF reasoning engine frequently uses these queries to compute values of transitive properties such as rdf:type. For the example query, we can select b and d together. A join-key, b preserves triple patterns among a, b, and c. Another join-key, d covers triple patterns among c, d, and e. In the next iteration, we can finally join them by selecting c or d as a join-key.

4.4.3 Hybrid Key Selection

Essentially, the greedy selection strategy and the multiple selection strategy can be used together. Let us consider the following chain-join example:

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D) \bowtie U(D,E)$$

In this example, we have three join keys $JK = \{B,C,D\}$ in total. Since the number of relations to be joined by a join key is the same, we first select a random join-key B. Then, relation R and S can be participated into a MapReduce job. At the same time, we can realize there still remains relations T and U. Therefore, we can select another join key D for the same MapReduce job. Then, the example join query can be represented as follow:

$$\{R(A,B) \bowtie S(B,C)\} \bowtie \{T(C,D) \bowtie U(D,E)\}$$

The result of first MapReduce job is $R \bowtie S$ and $T \bowtie U$. In the second MapReduce job, we can select a join key C so that we can generate final join results.

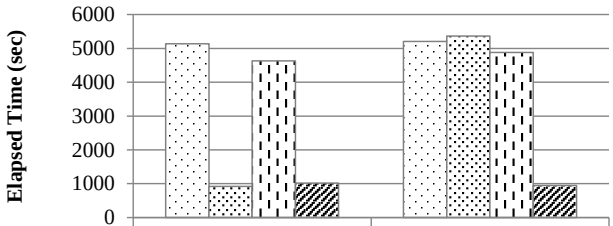
4.5 Experiments

In this section, we conduct experiments on skew handling algorithms over multi-way join queries. In Section 4.5.1, we first examine the performance of SK-Join queries with several skew handling algorithms. The experiment requires only a single MapReduce job. We vary the degree of skewness in each relation. In Section 4.5.2, we test another type of multi-way joins: MK-Join queries. Processing MK-Join queries require multiple MapReduce jobs. In each job, the elapsed time depends on the longest processing time of individual joins. Finally, in Section 4.5.3, we conduct a case study on multi-way joins in analyzing TV watch logs.

4.5.1 SK-Join Experiments

The first experiment is about SK-Join queries. By extending a partitioning matrix to a cube, our approach handles SK-Join queries in the same manner. As shown in Figure 32, we tested two extreme cases of join product skew: (x10 $\bowtie$ x10 $\bowtie$ x10M) and (x1K $\bowtie$ x1K $\bowtie$ x1K). Therefore, the second case has higher possibility of join product skew than the first case. Each relation has 100M input tuple, and the output size is over 10 billion tuples and 600GB. Since we have three relations to be joined, we use 27 cores for reducers because the RANDOM algorithm requires $k^{1/3}$ to be an integer.

Experimental results in Figure 32 reconfirm conclusions of the scalar



	x10⋈x10⋈x10M	x1K⋈x1K⋈x1K
HASH	5136	5209
RANGE	917	5366
RANDOM	4630	4880
MDR	1015	947

Figure 32: Performance on SK-Join queries

skew experiments. The RANGE algorithm is vulnerable to the presence of join product skew, and the RANDOM algorithm has a burden of input duplication. Actually, in star-join queries, the problem of input duplication becomes worse than that of the two-way join queries. To produce correct join results, the RANDOM algorithm has to duplicate all input tuples $27^{2/3} = 9$ times regardless of the presence of skew. We have 100M tuples in each relation, which means that 300M * 9 = 27B tuples are entered to the shuffle phase in MapReduce. Therefore, our approach outperforms the other approaches as shown in the experimental results.

However, it should be noted that our approach requires more time to lookup candidate cells. In *listCell* function of Algorithm 6, we has to examine k^3 cells in order to filter our non-candidate cells. On the other hand, RANGE just has a one-dimensional partitioning vector whose length is k, and RANDOM has a cube consisting of k cells. We guess this is a reason of that the RANGE algorithm is faster than ours in the (x10 ⋈ x10 ⋈ x10M) experiment. Even though the difference is not so significant, we think the

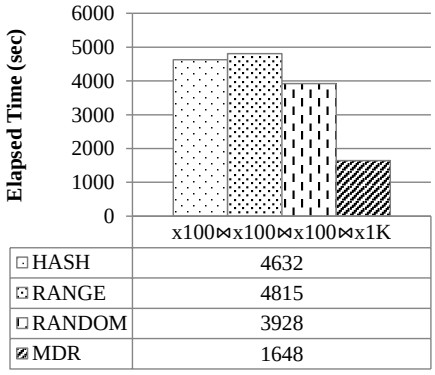


Figure 33: Performance on MK-Join queries

margin can become narrow if we adopt a spatial index structure [37] in order to reduce the lookup time.

4.5.2 MK-Join Experiments

The next experiment is about a MK-Join query. In this experiment, we compute a join $R \bowtie S \bowtie T \bowtie U$. Each relation has 100M input tuples and follows the scalar skew. Although the data schema (pk, jk) is the same so that we can compute the join as a star-join query, we intentionally process it with two MapReduce jobs. In the first job, $R \bowtie S$ and $T \bowtie U$ are evaluated simultaneously. We create $M_{1,1}(R,S)$ and $M_{1,2}(T,U)$ with samples from four relations. In the second job, we evaluate the join between intermediate join results.

It should be noted that we slightly modify original Map and Reduce functions in Algorithm 6 and 7. The difference is that we attach a matrix tag in addition to a relation tag. In repartition join, we add a relation tag to an input tuple in order to distinguish input relations in the Reduce function. In

two-way joins, a Reduce function receives input tuples from two relations. In multi-way joins, the number of input relations becomes larger, and we have to decide a join in which an input tuple contribute to. Therefore, for an input tuple r from R, we added $M_{1,1}$ tag and R tag. In the Reduce function, we create different build input tables according to the matrix tag.

Experimental results in Figure 33 shows the execution time of given chain-join query. As shown in other experiments, the MDRP algorithm outperforms the other skew handling algorithms.

4.5.3 Analysis of TV Watching Logs

In previous sections, we evaluate the effectiveness of our algorithm with synthetic data sets. However, there are a number of real-world data sets which require joins between skewed relations. In this section, we conduct a case study with a real-world data set.

Analyzing TV watching logs is closely related to the skew handling problem. Generally, people watch only a few channels on the television. In Korea, some public TV channels, such as 6 - 13, occupy over 70% of the broadcasting rating. Therefore, every analysis task incurs skewed execution times.

First of all, we simply describe our test data set. The data set contains four relations: User(U), Program(P), Channel(C) and Channel Category(CC). There are foreign key constraints between $U-P$, $P-C$ and $C-CC$. Although we have examined many test queries, we will report a result on a

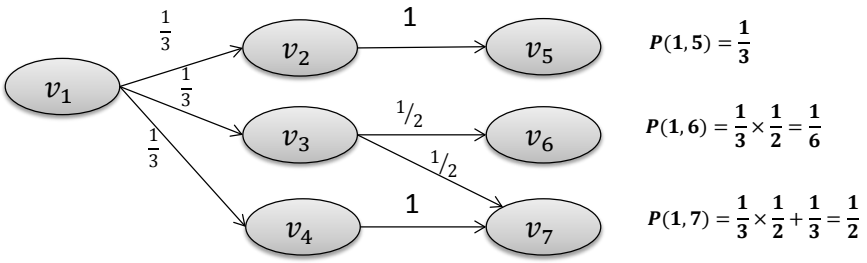


Figure 34: Graph analysis using matrix multiplication

representative test query which shows clear trends of data skewness:

$$U - P - U - P - C - CC - C$$

In above query, $U - P$ indicates programs p that have watched by a specific user u. $U - P - U$ means that users who have seen at least one the same programs with the specific user u, and similarly $U - P - U - P$ is a set of programs that have been watched by the similar users. Similarly, $C - CC - C$ represents a set of channels which belong to the same category. Therefore, the given query means that channels that belong to the same category watched by similar users for each user u.

The data set can be represented in a graph. Since we have four relations, we have four different entity types corresponding each relation. Relationships between entities can be described by the foreign key constraints. Then, with the matrix multiplication algorithm, the test query can provide us a recommendation result for each user in the graph. Let us suppose that there is a user corresponding to the v_1 in Figure 34. We now want to decide the most close node from v_1 among v_5, v_6, v_7. We first are able to decide

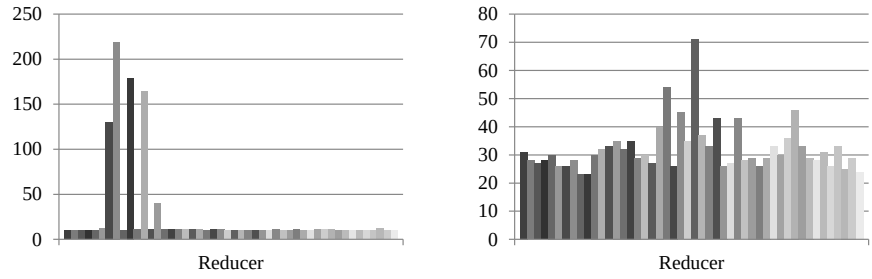


Figure 35: Performance comparison between w/ and w/o skew handling

weights of all edges according to the number of edges of each vertex. Then, we can compute the distance between v_1 and other candidate nodes as shown in the example. The vertex v_7 has gained the highest weight so we can draw a conclusion that v_1 and v_7 are close.

In our experiment, join operation between $U-P-U-P$ and $C-CC-C$ incurs skewed distribution of processing times among reducers. In the left of Figure 35, we show the elapsed time of each reducer. Some reducers spend most of processing times while the others are already finished. The maximum time is 219 seconds. On the other hand, the right figure shows that our algorithm moderates the difference of processing times across reducers. The maximum time here is only 71 seconds which is 1/3 compared to the previous case. This result shows that our skew handling algorithm is useful in real-world applications.

4.6 Summary

In this chapter, we addressed the problem of data skew in multi-way joins. Our proposed MDRP technique was extended to deal with SK-Join

queries and MK-Join queries which are sub types of multi-way joins. For SK-join queries, we extend the partitioning matrix to a multi-dimensional partitioning cube. For MK-join queries, we exploit two or more partitioning matrices (or cubes) in a MapReduce job.

Efficient processing of multi-way joins requires more sophisticated techniques beyond handling of data skew. Handling of iterative MapReduce job is one of the most important issues. We have analyzed costs of iteration-based and replication-based algorithms to process iterative jobs. In addition, we have proposed join-key selection strategies and algorithms for minimizing the number of MapReduce jobs.

Chapter V

Applications

In Chapter 4, we have already introduced two applications that require joins of multiple relations. We now discuss about detail implementations and experimental evaluations for those applications. In Section 5.1 and 5.2, we show how our techniques can be integrated with algorithms for graph pattern matching and matrix chain multiplication problems.

5.1 Algorithms for SPARQL Basic Graph Pattern

As shown in Section 4.1.1, the graph pattern matching problem is an important application of multi-way joins. In this section, we offer general and efficient MapReduce algorithms in addition to our skew complex join processing algorithm in order to handle entire basic graph patterns. Especially, we focus on two aspects of the problem. First, in a MapReduce world, it is known that the join operation requires computationally expensive MapReduce iterations. We minimize the number of iterations with the followings: 1) We adopt traditional multi-way join into MapReduce instead of multiple individual joins. 2) By analyzing a given query, we select a good join-key to avoid unnecessary iterations. Second, each join operation exploits the skew handling algorithm proposed in the previous chapter. As a

result, the algorithm shows good performance and scalability in terms of time and data size.

The main contribution of this section is that we offer an efficient multi-way MapReduce algorithm for processing SPARQL BGP queries. For queries having two or more join-key candidates, we use two join-key selection strategies introduced in Section 4.4. These strategies have the effect of reducing the number of unnecessary MapReduce jobs. An experiment with the Lehigh University Benchmark (LUBM) [38] shows that the algorithm provides scalable access to the RDF data.

Most SPARQL queries contain a set of triple patterns known as a Basic Graph Pattern. Triple patterns are similar to RDF triples (s, p, o) except that each of the subject, predicate and object can be a variable. We assumed that RDF triples are stored in a N-Triples format file. To evaluate a BGP, we offer two separate MapReduce operations: namely MR-Selection and MR-Join. MR-Selection obtains RDF triples which satisfy at least one triple pattern and MR-Join merges matched triples into a matched graph. MR-Join can be performed iteratively when a BGP has two or more shared variables.

5.1.1 MR-Selection

MR-Selection obtains RDF triples which satisfy at least one triple pattern. In addition, MR-Selection projects a given triple onto the satisfied triple pattern in order to assign a value to a corresponding variable. The input format of MR-Selection is simple. The algorithm receives a (s, p, o) triple expressed in N-Triple format. On the other hand, the output format of the algorithm is quite complex. An output result, which is in a pair of

Input: (<Prof0>, rdf:type, ub:Professor)
(<Prof0>, ub:worksFor, <Department0>)
(<Prof0>, ub:name, "Professor0")
(<Prof0>, ub:emailAddress, "prof0@email.com")
(<Prof0>, ub:telephone, "000-0000-0000")
(<Prof1>, rdf:type, ub:Professor)
(<Dept0>, ub:name, "Department0")
…

Output: (<1>x, [x]Prof0) (<1>x, [x]Prof1)
(<2>x, [x]Prof0) (<3>x|y1, [x]Prof0|[y1]Professor0)
(<4>x|y2, [x]Prof0|[y2]prof0@email.com)
(<5>x|y3, [x]Prof0|[y3]000-0000-0000)

Figure 36: An example of MR-Selection

parentheses, can be divided into a key part and a value part separated by a comma. The key part consists of a satisfied triple pattern number and variables included in the triple pattern. The value part can be divided by vertical bar characters; each subpart consists of a variable name and a corresponding value.

With a SPARQL query example in Figure 19, let us consider example inputs and outputs of MR-Selection as shown in Figure 36. In this example, a result of an input triple (<Prof0>, rdf:type, ub:Professor) is (<1>x, [x]Prof0). A triple (<Dept0>, ub:name, ”Department0”) will be filtered out because the triple does not satisfy any triple patterns.

Map: (k1, v1) -> [(k2, v2)]	Reduce: (k2, [v2]) -> [(k3, v3)]

function map {
  read input triple // example triple: s1 p1 o1
  for each(triple pattern in a query){
  // BGP example = {
  // 0:?a p1 ?c. 1:?a p2 ?b. 2:?b p1 o1
  // }
 	
  if(input triple satisfies at least one of given triple patterns){
  make key and value
  /*
  key = a concatenated string of (variable name, value)
  value = # of the satisfied triple pattern
  key = ([a]s1|[c]o1), value = 0
  key = ([b]s1), value = 2
  */
  output(key, value)
  }
  }
}
	function reduce {
	  read inputs
	  /*
	  reducer1: key = ([a]s1|[c]o1), values = 0
	  reducer2: key = ([b]s1), values = 2
	
	  values to have the same map-key are mapped into the same reducer
	  */
	
	  for each(value in values){
	  make key and value
	  /*
	  reducer1: key = (<0>a|c), value = ([a]s1|[c]o1)
	  reducer2: key = (<2>b), value = ([b]s1)
	  */
	  output(key, value)
	  }
	}

Figure 37: MR-Selection (Pseudo-code)

Input: (<1>x, [x]Prof0) (<1>x, [x]Prof1)
(<2>x, [x]Prof0) (<3>x|y1, [x]Prof0|[y1]Professor0)
(<4>x|y2, [x]Prof0|[y2]prof0@email.com)
(<5>x|y3, [x]Prof0|[y3]000-0000-0000)

Output: (<1|2|3|4|5>x|y1|y2|y3, [x]Prof0|[y1]Professor0|[y2]
prof0@email.com|[y3]000-0000-0000)

Figure 38: An example of MR-Join

Figure 37 explains the algorithms of the Map and Reduce functions. The Map function filters unnecessary triples out, and the Reduce function creates the final results according to the output format. Conceptually, the MR-Selection algorithm produces temporal tables which satisfy each triple pattern. A result table has variable names as a relational table has attribute names. It also has values for the variable names, as does the relational table. The resulting table will be used for the next MR-Join operation if necessary.

5.1.2 MR-Join

MR-Join merges matched triples or partially matched graphs into a matched graph. The output of MR-Selection is delivered to MR-Join if a BGP contains one or more shared variables. If a BGP has no shared variables, there is no reason to proceed to the next MR-Join operation. MR-Selection and MR-Join can be connected because the input format of MR-Join is identical to the output format of MR-Selection. The output format of MR-Join is also identical to the output format of the MR-Selection algorithm. It is notable that the same data format enables the MR-Join iteration.

The input and output examples in Figure 38 show the role of MR-Join. As with MR-Selection, MR-Join has both Map and Reduce functions. The Map function of MR-Join creates separate maps according to the join-key variable and the corresponding value. Suppose that variable x is used as a join-key. In this example, there will be two separate maps because the inputs have two distinct value [x]Prof0 and [x]Prof1 as a join-key. The Reduce function for MR-Join merges inputs in a given map. A reducer for a map created by [x]Prof0 generates a merged output, as expressed in the example. On the other hand, a reducer for a map created by [x]Prof1 cannot generate an output because the inputs in the map do not satisfy all triple patterns related to a variable x. Figure 39 shows how the MR-Join algorithm works in detail.

Map: (k1, v1) -> [(k2, v2)]

```
function map {
    read input
    // example: ((<1>a|b), ([a]a1|[b]b1)), ((<3>a|c), ([a]a1|[c]c1))

    split the input
    // triple pattern number (tpn) = 1, 3
    // variable name (vn) = a, b, c
    // variable name & value (vv) = ([a]a1|[b]b1), ([a]a1|[c]c1)

    get a set of join key(mkey_vn) and corresponding triple pattern
    numbers(mkey_tpn) to be satisfied

    for each(mkey_vn determined by BGP analyzer){
      if(mkey_vn is in vn && mkey_tpn contains tpn) {
        // example: mkey_vn = a, mkey_tpn = [1, 3]

        key = mkey_vv          // key = [a]a1
        value = (tpn, vv)      // value = (<1>, [a]a1|[b]b1))
        output(key, value)
      }
}
```

Reduce: (k2, [v2]) -> [(k3, v3)]

```
function reduce {
    read inputs
    // key = [a]a1
    // values = [(<1>, [a]a1|[b]b1), (<3>, [a]a1|[c]c1)]

    get a set of join key(mkey_vn) and corresponding triple pattern
    numbers(mkey_tpn) to be satisfied

    for each(value in values){
      make a temp hashtable H
      // a key for H: (tpn, vn), values for H: [(vv)]
      // example H = {((<1>,a|b):[(a1|b1)]), ((<3>,a|c):[(a1|c1)])}
    }

    if( (tpn,(mkey_vn)) is the same with the key in H ) {
        make a Cartesian product among values in H
        // (a1,b1), (a1,c1) -> (a1, b1, c1)
        output(key, value)
    }
}
```

Figure 39: MR-Selection (Pseudo-code)

5.1.3 Performance Evaluation

In this section, we demonstrate performance of our approach. The experimental results show that our proposed algorithm benefits from the multi-way join technique.

5.1.3.1 Experimental Setup

As mentioned earlier, we used LUBM [38] for all experiments to assess the performance and the scalability of our approach. LUBM offers a synthetic data generator and test queries for evaluating SPARQL query processors. The generator produces OWL [39] format data that contains the people and activities of a university. We used the Jena [40] framework to pre-compute the transitive closure expressed in the OWL document because the goal of our research is RDF graph pattern matching rather than OWL reasoning. We first created 6 datasets with 1, 5, 10, 25, 50, 100 universities, respectively, with random seed 0. In this experiment, LUBM (n) indicates that the dataset contains the n universities.

We built an environment using Amazon Web Services (AWS). Cloudera's Hadoop Distribution (CDH) was used on the Amazon EC2 (1.7GB of RAM, 5 EC2 Compute Units). The Hadoop Distributed File System (HDFS) was built on the Amazon Elastic Block Store.

5.1.3.2 Performance Evaluation

We first show the effect of multi-way join algorithms. Table 6 shows the execution times for all test queries. Multi-way join technique reduces

the execution time by joining several triple patterns at once. However, some queries do not show a significant difference because they are too simple to take advantages of multi-way joins. In addition, Table 7 shows the execution times for all queries and data size. While we increase the data size, the algorithm shows scalable execution times.

Table 6: A comparison of join algorithms

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
2-way	123.4	181.6	69.8	256.6	75.5	44.1	205.6	232.5	256.0	68.8	66.8	112.8	73.3	47.1
m-way	86.4	104.0	67.2	126.4	74.1	44.5	135.0	140.4	152.7	73.3	63.5	86.1	72.8	42.1
diff.	36.9	77.5	2.5	130.1	1.3	-0.3	70.5	92.1	103.2	-4.5	3.2	26.6	0.5	4.9

Table 7: Execution time for the LUBM benchmark

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14
LUBM(1)	44.4	65.5	42.6	47.3	42.2	20.0	66.5	64.1	66.0	43.3	42.6	65.8	45.0	19.8
LUBM(5)	54.6	72.8	47.7	52.3	50.6	26.0	77.7	75.2	75.0	53.6	53.4	76.8	52.1	26.0
LUBM(10)	48.7	71.5	53.3	60.1	49.3	27.0	73.0	88.9	74.6	48.6	54.4	76.4	49.6	26.2
LUBM(25)	63.6	83.1	57.7	74.8	55.5	33.8	99.1	93.8	98.9	53.5	52.4	80.9	57.9	34.0
LUBM(50)	71.4	90.9	63.3	88.9	60.6	35.9	103.3	107.7	108.2	63.6	52.4	84.9	64.6	36.2
LUBM(100)	86.4	104.0	67.2	126.4	74.1	44.5	135.0	140.4	152.7	73.3	63.5	86.1	72.8	42.1

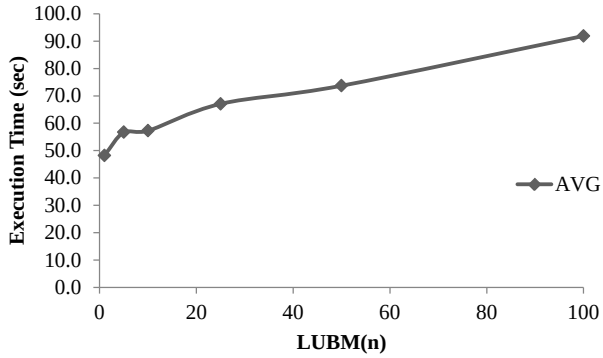


Figure 40: Average execution time of LUBM queries

Figure 40 represent the average execution times for each data size. The average execution time for LUBM(1) is 48.2 seconds, while LUBM(100) requires 91.9 seconds to finish the request. Hence, the algorithm effectively utilizes the computing resources. The execution time is increased only 2 times while the number of triples is increased 100 times. One likely reason for this result is the format of the map-key used in the MR operations. A map-key contains a variable name and a corresponding value for the variable. A number of distinct values produce a number of reducers, which implies that the MR framework effectively utilizes distributed resources. If we use only the variable name, the number of reducers is decreased, which indicates that the most of triples are skewed to a reducer. The MR framework attempts to assign a reducer to an independent node in order to enable massive parallel processing. Consequently, it is a reasonable implementation that the number of reducers is greater than the number of nodes.

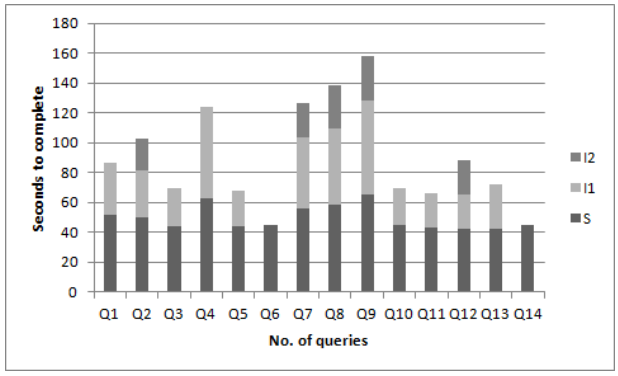


Figure 41: Time used for MR-Selection and MR-Join

5.1.4 Discussion

Several implementation issues remain which may have an effect on the scalability and the performance of the algorithm. In this subsection, we deal with some of the issues related to this.

Selection Optimization using Indexes: The execution times of the algorithm in actuality appear to be slow. We can apply some optimization techniques to improve the performance of the algorithm. First, we considered that the MR-Selection algorithm can be a bottleneck because we do not use any indexes. Therefore, MR-Selection must examine all of the RDF triples in a distributed file system whenever a SPARQL query is issued. Figure 41 supports that the MR-Selection should be managed with specific optimization techniques. It shows the individual execution time spent by MR-Selection and MR-Join operations. Some complex queries have several join iterations, but most of the test queries require a MR-Selection and a MR-Join operation. The MR-Selection should be verified because it requires more

than 40 seconds for all test queries. For this reason, the construction of an index using HBase can be considered [41]. HBase is a distributed storage system that runs on the HDFS file system. The index structure is simple in this case. For a triple (s, p, o), s is employed as a row key and p is used for a column key. The value of a cell is o. If we have several object values for a certain cell, we can use timestamps to distinguish values. We may be able to exploit the exhaustive indexes introduced in two related studies [42] and [43].

However, we have to consider that it is a very time-consuming task to load RDF data into HBase. As mentioned in earlier research [44], for some of the benchmarks, it is possible to run 50 separate MR analyses over the data before the data can be loaded into a database and a single analysis completed. As an example, an earlier research, [45] takes 16 minutes to load 100 million triples. Hence, we have to consider the purpose of the data analyses.

MR Pipelining: One of the features of the MapReduce framework is the writing of intermediate results into a file system. In other words, the framework materializes all intermediate data. Thus, the execution time of a MapReduce job is relatively slower compared to other approaches. Particularly, MR-Join occasionally requires iterative processing, which passes data through the disk I/O and the network data transfer. To improve the performance of MapReduce jobs, two methods were recently reported [46] and [47]. They transfer intermediate data to the next step while the previous step is still running. This improves the overall performance.

However, it is controversial to use MapReduce pipelining for large-scale fault-tolerant data analysis. The MapReduce framework achieves fault-tolerance due to materialization and replication. Clearly, there is a trade-off between performance and reliability. Hence, this trade-off should be considered carefully. For an accurate calculation, reliability is more important than performance.

Dictionary Encoding: In this paper, we used raw text instead of an encoded string. For example, we use a join-key format such as '[variable]value'. However, as mentioned in earlier research [44], it is inefficient to parse the ad-hoc format.

In this section, we have proposed a MapReduce algorithm for BGP processing. We also discussed various aspects of the algorithm with experimental results. In a comparison with an existing approach, the algorithm showed superior performance. Although there are several implementation issues remaining, the algorithm reduces the number of join iterations, which has a considerable effect on the overall execution time.

5.2 Algorithms for Matrix Chain Multiplication

In this section, we address the matrix chain multiplication problem introduced in Section 4.1.2. Although several studies have investigated the problem [48, 49, 50, 51], most of them have focused on the efficiency of a binary multiplication. For example, suppose that we evaluate the multiplication of three matrices, i.e. $M_1 \times M_2 \times M_3$. The previous studies solve this problem with sequential (pipelined) multiplications of two matrices,

$(M_1 \times M_2) \times M_3$. Therefore, they focus on an efficient parallel algorithm for an individual binary multiplication. However, we adopt the multi-way join operations into the problem [52].

Actually, the join-based matrix multiplication $\bowtie_*$ is associative, as is the matrix multiplication. Therefore, the following multiplications are all equivalent, when we have matrices A, B, C, and D.

$$\begin{aligned} A \bowtie_* B \bowtie_* C \bowtie_* D &= (((A \bowtie_* B) \bowtie_* C) \bowtie_* D) \\ &= ((A \bowtie_* B) \bowtie_* (C \bowtie_* D)) \\ &= (A \bowtie_* B \bowtie_* C \bowtie_* D) \end{aligned}$$

There are a number of ways to compute a matrix chain multiplication. A typical method is to use sequential multiplication, as described in the first equation. We refer to this as serial two-way join (S2). In S2, each $\bowtie_*$ operation requires a separate MapReduce job. Although S2 employs the concept of parallelism, it is limited in its operation, i.e., it involves intra-operation parallelism. Another way to evaluate the above expression is through a parallel two-way join operation (P2). This approach follows inter-operation parallelism, which means that we can compute $A \bowtie_* B$ and $C \bowtie_* D$ simultaneously. We can expand the concept of inter-operation parallelism so as to compute the matrix chain with an MapReduce job, as represented in the last equation. We refer to this approach as a parallel m-way join operation (PM). In this section, we explain our implementation of these operations with the MapReduce framework.

A MapReduce job incurs considerable cost in terms of computation

time. Whenever a MapReduce job is launched, the job tracker makes copies of the program for all task nodes. The task nodes read data from the distributed file system and write the result after the job is finished. When iteration of the job occurs, unnecessary cost will be incurred again for the described tasks. Therefore, exploiting inter-operation parallelism reduces the number of MapReduce job iterations, leading to improvements in the algorithms.

5.2.1 Serial Two-Way Join (S2)

In S2, we implemented the improved repartition join as described in Section 2.4. Because we are focusing on the matrix chain multiplication process, the *driver* function, which controls the sequence of MapReduce jobs, is important. In this section, we explain the *driver* function; pseudo-code for the *map* and *reduce* functions are outlined in the literature [5]. As described in Algorithm 8, S2 computes the join operation between the first two relations, and then produces the next join results between the previous result relation and the next relation. A join operation requires a MapReduce job.

Algorithm 8 Driver function for serial two-way join

Input: Relations $M_1, M_2, ..., M_n$ representing matrices

1: $M_{left} = M_1$
2: **for** $i = 2$ to n **do**
3: $\quad M_{right} = M_i$
4: $\quad M_{result} =$ doMR-S2 (M_{left}, M_{right})
5: $\quad M_{left} = M_{result}$
6: **end for**

It should be noted that a binary multiplication $\bowtie_*$ can be separated

into two MapReduce jobs: a job for the join operation and another job for a sum of join results. For example, consider two matrices A and B, with $i \times k$ elements in A and $k \times j$ elements in B. The first MapReduce job computes $A_{ik}B_{kj}$ for all i, j and k. This can be done by the join MapReduce job as described in Alg. 8. However, we need the second job that computes $C_{ij} = \Sigma_k A_{ik}B_{kj}$. Here, we present an algorithm that yields a join result for several relations at once. Then, the sum of the join result has to be equal to the join result of serial two-way joins.

Theorem 3. *The sum of the multi-way join result is equal to the sum of the join result of serial two-way joins.*

$$((A \bowtie_* B) \bowtie_* C) = (A \bowtie_* B \bowtie_* C)$$

Proof. According to relational algebra, equation (4) is true when $\Sigma(\Sigma(A \bowtie B) \bowtie C) = \Sigma(A \bowtie B \bowtie C)$. It is easy to see that the above equation is true, because the join operation takes only elements sharing the same join key. In other words, $((R_1 \bowtie R_2) \bowtie R_3)$ is equal to $(R_1 \bowtie R_2 \bowtie R_3)$, so the position of the sigma does not affect the final result. For example, suppose that $D = A \bowtie_* B \bowtie_* C$. Then, $d_{11} = ((a_{11}b_{11} + a_{12}b_{21})c_{11} + (a_{11}b_{12} + a_{12}b_{22})c_{21}) = a_{11}b_{11}c_{11} + a_{12}b_{21}c_{11} + a_{11}b_{12}c_{21} + a_{12}b_{22}c_{21}$. This shows that the sum of join results among several relations is equal to the join result of serial two-way joins. □

5.2.2 Parallel M-Way Join (P2, PM)

Several studies, such as [53], [49], and [54], have examined binary multiplication between two matrices using the MapReduce framework in order to achieve intra-operation parallelism. However, we focus here on inter-operation parallelism. Therefore, the *driver* function of PM must be implemented in a different way.

Algorithm 9 Driver function for parallel m-way join

Input: Relations $M_1, M_2, ..., M_n$ representing matrices
Input: An integer m indicating m-way join

```
1:  LIST_M_next ⇐ [M_1, M_2, ..., M_n]
2:  while |LIST_M_next| > 1 do
3:     for i = 1 to |LIST_M_next| do
4:        if (i mod m) == 1 then
5:           add M_i to LIST_M_left
6:           M_left = M_i
7:        else
8:           add M_i to LIST_M_right(M_left)
9:        end if
10:    end for
11:    LIST_M_next = doMR-PM (LIST_M_left, LIST_M_right)
12: end while
```

Algorithm 9 shows pseudo-code for the *driver* function for a parallel m-way join. It should be noted that the parallel two-way join is a special case of the m-way join, where m is 2. In the algorithm, we prepare a list of matrices for a MapReduce job. We refer to this list as $LIST\_M_{next}$. Every matrix will be included on the list for the first job. We then add every $(m+1)$-th matrix to another list, $LIST\_M_{left}$. Every matrix in $LIST\_M_{left}$ has a list of matrices to be multiplied. We denote these individual lists for M_{left} as $LIST\_M_{right}(M_{left})$. As a result, a MapReduce job can compute all

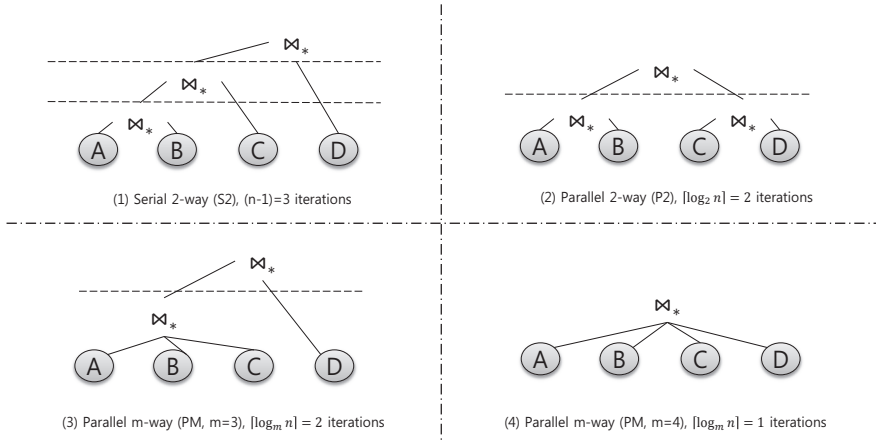


Figure 42: Required number of MapReduce jobs

multiplications for each left matrix in the list $LIST\_M_{left}$. The result will be passed to the next MapReduce iteration, when the number of matrices in the $LIST\_M_{next}$ is greater than 1.

The multi-way join algorithm reduces the number of MapReduce jobs. In the case of the S2 algorithm, we need $(n-1)$ iterations where n is the number of matrices. Figure 42 shows an example. We have four matrices from A to D. S2 requires $3 = 4-1$ MR jobs because it takes only two matrices in a job. However, If we use the P2 algorithm, the number of MR jobs is reduced to $\lceil \log_2 n \rceil$. The P2 algorithm takes all matrices as inputs of a MR job and produces $\lceil n/2 \rceil$ intermediate result matrices. In Figure 42(2), $A \bowtie_* B$ and $C \bowtie_* D$ are computed within a MR job. This reduces the total number of MR jobs. Similarity, if we set $m = 3$, a MR job produces $\lceil n/3 \rceil$ intermediate result matrices and finally requires $\lceil \log_3 n \rceil$ MR jobs.

Theorem 4. *Let n be the total number of matrices and m be an integer*

Algorithm 10 Map function for parallel m-way join

Input: $LIST\_M_{left}$ from the driver function
Input: An integer m indicating m-way join
Input: An input record $r = (tag, row, col, val)$

```
1: if m == 2 (parallel two-way join) then
2:    if r comes from matrices in LIST_M_left then
3:       output (r.col, r)
4:    else
5:       output (r.row, r)
6:    end if
7: else
8:    K ⇐ a list of composite keys (i.e. identifiers for reducers)
9:    for k in K do
10:      output (k, r)
11:   end for
12: end if
```

that indicates the m-way joins. The number of MapReduce jobs for the PM algorithm is $\lceil \log_m n \rceil$.

Proof. The first MR job takes all matrices as its inputs and produces $\lceil n/m \rceil$ number of intermediate result matrices. The second MR job takes the intermediate results and produces $\lceil \lceil n/m \rceil / m \rceil$. This process is repeated until the number of intermediate result matrices is equal to one. Therefore, $n < m^x$ is valid where the x is the total number of MR jobs. □

There are several ways to implement the *map* and the *reduce* functions of the multi-way join operation in terms of the join key construction. We use different startigies for P2 and PM, as demonstrated in Algs. 10 and 11. In the case of P2, we can use a raw key. Suppose that we have matrices $M_1(r_1, c_1, v_1)$,$M_2(r_2, c_2, v_2)$, and $M_3(r_3, c_3, v_3)$. Then, the first MR job matches $M_1.c_1 = M_2.r_2$ and produces the records $(r_1, c_2, v_1 * v_2)$. In the rela-

Algorithm 11 Reduce function for parallel m-way join

Input: $LIST\_M_{left}$, $LIST\_M_{right}$, from the driver function
Input: R, a list of input record (tag, row, col, val) sorted by the tag

```
1: cur ⇐ a cursor for a list R
2: for left in LIST_M_left do
3:    if |LIST_M_right(left)| == 0 (a leftover matrix for the next iteration)
      then
4:       output the current matrix R[cur] for the next MR job
5:       continue
6:    end if
7:    if |LIST_M_right(left)| == 1 (the two-way join) then
8:       create partitions for the left matrix R[cur]
9:       do a hash-join between the left matrix and the current right matrix
         R[cur]
10:   end if
11:   for right in LIST_M_right(left) (the m-way join) do
12:      do serial two-way hash-joins in a local machine
13:   end for
14: end for
```

tional algebra, this is denoted by the natural join. We can use the raw values of c_1 and r_2 without any modification, meaning that the join key is "raw".

On the other hand, PM has to consider M_3 in a MapReduce job. There should be two different join keys for $M_1 \bowtie_* M_2$ and $M_2 \bowtie_* M_3$. Therefore, PM uses a composite key rather than a raw key. For example, A join key for M_1, M_2, and M_3 is $(c_1 = r_2, c_2 = r_3)$. If a mapper of PM takes a record (r_2, c_2, v_2) from M_2, then it can generate a single join key (r_2, c_2). However, the use of a composite key leads to record duplications. If the mapper takes a record (r_1, c_1, v_1) from M_1, then it can know only a part of the join key $(c_1, *)$. Therefore, we have to duplicate the record as if the $*$ represents all possible values of r_3. More detail explanations of the construction method regarding the number of machines is described in [13]. Although

the composite key enables PM to reduce the number of MR jobs, the use of a composite key leads to record duplications, with results of the communication overhead. In the next subsection, we will discuss other limitations of the composite key.

5.2.3 Serial Two-Way vs. Parallel M-Way

In the previous subsection, we see that there is a trade-off between iteration-based algorithms and the replication-based algorithm. Iteration-based algorithms require high I/O costs whereas the replication-based algorithm needs high communication cost.

This analysis support that the hybrid of iteration-based and replication-based algorithm is a good choice for many normal join queries. Our greedy-key-selection strategy is based on iteration-based algorithms. There is no duplication in the processing steps of greedy-key-selection algorithm. However, when given join operation is a *pure* chain-join, the greedy-key-selection algorithm is equal to cascade of 2-way joins which requires several MapReduce iterations. This is the reason why we need the multiple-key-selection strategy. Since the MapReduce framework has the key-equality based data flow, we can distinguish input tuples from different relations.

In addition, even in a *pure* chain-join, we can use a hybrid approach between the iteration-based approach and the replication-based approach. PM shows an example. The matrix chain multiplication problem contains the pure chain-join. But we can compute m relations in a join unit using replication-based algorithm. Then, the multiplication of n relations can be produced by only $log_m n$ MapReduce jobs. Moreover, we can reduce the I/O

costs between iterative jobs. We show the idea in the next subsection.

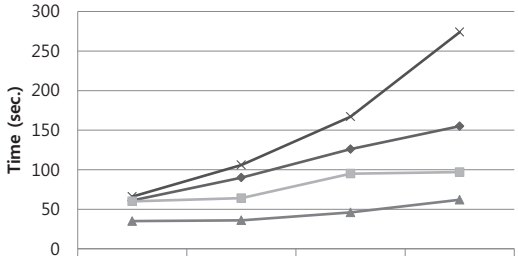
5.2.4 Performance Evaluation

We now present the performance results of the different algorithms described in this section. We built an 8-node Hadoop (v1.1.0) [2] cluster, with each node having an Intel i3-2100 dual-core CPU with 3.10 GHz speed and 4 GB memory. We used graph data sets from the Stanford Large Network Dataset Collection [55] to create an adjacency matrix. Specifically, we used 'p2p-Gnutella04.txt', 'amazon0302.txt', 'roadNet-PA.txt' files. The size of data sets is specified in Figure 44.

5.2.4.1 Efficiency of the Parallel M-way Join

Our first experiment is concerned with the efficiency of the parallel m-way join algorithm. We created an adjacency matrix of a graph dataset containing about 10,000 vertices, from which we computed powers of the matrix. For comparisons with state-of-the-art, we employed the Hive system (v0.9.0, released in 2012) [56]. The system provides users with a SQL interface for the MapReduce-based data management. Therefore, the S2 algorithm can be implemented with the Hive system as we can see the SQL query in Section 4.1.2.

The result is described in Figure 43. In the cube of M, the baseline (HIVE) and our implementations (S2, P2, and PM) are equivalent. However, our algorithms outperform the baseline when the number of input matrices become large. The Hive, S2 and P2 algorithms both take about 60 s



	M^3	M^4	M^5	M^6
HIVE	66	106	167	274
S2	61	90	126	155
P2	60	64	95	97
PM	35	36	46	62

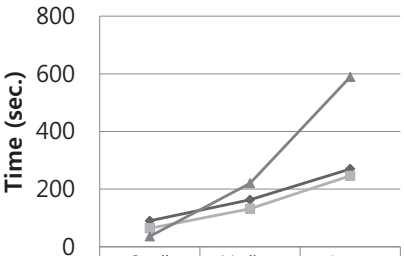
Figure 43: Execution time for computing powers of a matrix

to compute M^3 because they need the same number of MR jobs and use the same raw key. On the other hand, PM requires only 35 s. This result summarizes that reducing the number of MR iterations is indeed important. As the degree of powers increases, the gap between S2 and PM gradually grows. Even P2 outperforms the S2 algorithm. This result shows the importance of the inter-operation parallelism approach.

5.2.4.2 Limitations of the Parallel M-way Join

After the first experiment with small matrices, we increased the matrices' size. Figure 44 shows the experimental results. We used three data sets as described above. This result shows that the computation time for the PM algorithm is rapidly increased. Potential reasons for this are given below.

- As described in section 5.2.2, the parallel m-way join operation duplicates a (row, col, val) record according to the number of machines. The record duplication leads to a higher network cost between map-



	Small	Medium	Large
S2	90	163	270
P2	64	132	246
PM	36	220	589

M^4	Node	Edge
Small	10,876	39,994
Medium	262,111	1,234,877
Large	1,088,092	3,083,796

Figure 44: Trade-off between disk I/O and network overhead

pers and reducers.

- The composite key implementation of PM sacrifices intra-operation parallelism. In S2, we have a number of values for a join key. However, the maximum number of values for a join key in PM is the number of machines. Therefore, S2 has a number of partitions, each with a small number of records, while P2 has a small number of partitions, each with a large number of records. Moreover, Hadoop's mapper sorts local $(key, value)$ pairs according to their keys. Because PM generates many records for a key, its sorting overhead is larger than that of S2.

Although PM reduces the number of MR iterations, the cost of the shuffle phase of PM is higher than that of S2. P2 implements the raw key, so it still shows good performance for a large dataset.

5.2.5 Discussion

These experimental results show that we have to resolve the job iteration and network overhead. Implementing PM with the raw key will lead to the best performance. To do this, we can change the underlying record representation, thereby reducing the number of duplications and increasing the diversity of the join key within a MapReduce job. However, the scalable algorithm requires the sparse matrix representation scheme (row, col, val), which means that we cannot change the underlying record representation. Therefore, we need another approach to improve the performance of the algorithms. According to the experimental results, the parallel two-way join algorithm balances the inter-operation parallelism and the intra-operation parallelism approaches, because it can still use the raw key implementation. As a result, P2 shows the best performance for a large dataset. In the next section, we discuss how to adopt the P2 algorithm appropriately into the existing Hadoop infrastructure.

5.2.6 Extension: Embedded MapReduce

As we have shown through the experiments in Section 5.2.4, exploiting inter-operation parallelism is helpful for matrix chain multiplication. However, the existing Hadoop MapReduce framework is not sufficient to leverage the inter-operation parallelism for a chain of matrix multiplications. Data is stored in a shared storage. Once a MapReduce job is launched, data is split and delivered to mappers. The map function emits (k, v) pairs that will be sorted locally according to k. In the shuffle phase, (k, v) pairs

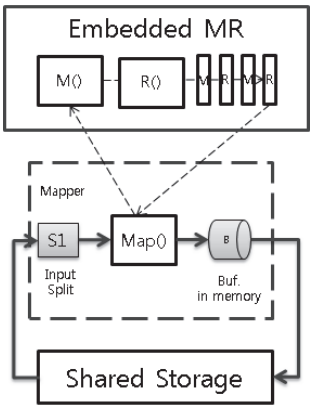


Figure 45: Embedded MapReduce jobs in a map-only job

are transferred to reducers and sorted again. After the reduce function, the results will be written back to the shared storage.

This processing flow has some demerits in optimizing the P2 algorithm. First, an iterative MapReduce job needs unnecessary disk I/O on the shared storage. Sorting in a mapper and a shuffle phase is also unnecessary. As a result, we need to modify the existing Hadoop implementation.

Our modification idea is a map-only job. As demonstrated in Figure 45, the Hadoop MapReduce framework allows a map-only job to avoid sorting in a mapper or the shuffle phase. Therefore, we can optimize the P2 algorithm if we can simulate the MapReduce framework inside a map task.

To implement the idea, we adopt the Apache Giraph framework [57]. This framework was started as an open-source version of Google's Pregel [58] which follows the BSP (Bulk Synchronous Parallel) model [59] for large-scale graph processing. One of important aspects of this framework is that it helps us to launch a map-only job on the existing Hadoop infrastruc-

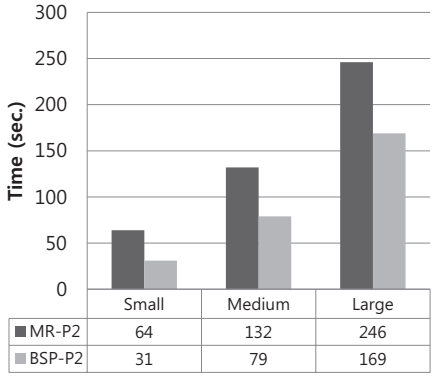


Figure 46: Execution time of BSP-based implementation

ture easily. We reimplement our P2 algorithm in order to use the map-only job feature.

It should be noted that the BSP model is convertible into the MapReduce programming model, a result proven by [60]. Therefore, a MapReduce job can be implemented with a series of supersteps and vice versa. Alg. 12 shows a detailed implementation of the P2 algorithm within the BSP framework. It can be divided into three parts. In the first superstep, we simulate the *map* function of the MapReduce algorithm. The second part plays the role of the *reduce* function. It computes join results between two matrices with a local hash-join algorithm. The last part aggregates the join results.

Figure 46 shows a comparison result on the efficiency. BSP-based P2 outperforms MR-based P2 because of reduced overhead between MapReduce job iterations. This is because the BSP-based implementation reduces unncessary disk I/Os. Consequently, our MapReduce-based algorithms can be adopted into existing Hadoop clusters with significant improvement in performance.

Algorithm 12 *Compute* function for P2 in the BSP framework

Input: $LIST\_M_{left}$, $LIST\_M_{right}$, from the driver function
Input: r, an input vertex value, (tag, row, col, val)

```
1:  // the map() function in P2
2:  if getSuperstep() == 0 then
3:     if LIST_M_left contains r.tag then
4:        sendMsg((r.tag, r.col), r)
5:     else
6:        sendMsg((r.tag, r.row), r)
7:     end if
8:     removeVertex(this)
9:  end if
10:
11: // the reduce() function in P2
12: if getSuperstep() == 1 then
13:    computeJoin()
14:    for each join result (left, right) do
15:       sendMsg((left.row, right.col), left.val * right.val)
16:    end for
17:    removeVertex(this)
18: end if
19:
20: // compute sum() for join results
21: if getSuperstep() == 2 then
22:    sum ⇐ 0
23:    while msgIterator.hasNext() do
24:       sum+ = msgItertor.next()
25:    end while
26:    voteToHalt()
27: end if
```

Chapter VI

Conclusion

Since the size of data to be processed is increasing dramatically, the MapReduce frameworks has gained attention recently. Although many data analysis tasks benefit from the framework, processing of n-way operation is still difficult. Joins are fundamental n-way operations which integrate different data sources.

There have been a number of studies on the join processing in MapReduce. In this paper, we added a new study on the effects of data skew in join algorithms using the framework. Specifically, we proposed a new skew handling technique, called Multi-Dimensional Range Partitioning, for efficient processing of parallel joins.

The proposed technique is more efficient than previous skew handling techniques, range-based and randomized partitioning techniques. When a join operation has join product skew, our algorithm outperforms the range-based algorithms. When the size of input relation is sufficiently large, our algorithm outperforms the randomized algorithms.

The proposed technique is scalable. Regardless of the size of input data, we can create sub-ranges that can be fit in memory. Moreover, the execution time of a join operation can be reduced as we add more machines into the cluster. On the other hand, the randomized algorithm produces more intermediate results when we increase the number of processing units.

In addition, the proposed technique is platform-independent. Although we examine our algorithms with the MapReduce framework, the MDRP technique itself can actually work with traditional parallel DBMSs. The range-based approach already used in many shared-nothing systems. Our algorithm improves the original range-based approach when we need to consider join results.

Finally, the proposed technique is applicable to several join types such as theta-joins and multi-way joins. We have demonstrated the effectiveness of our technique with extensive experiments on many synthetic and real-world data sets.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[2] "Apache hadoop. http://hadoop.apache.org."

[3] C. Doulkeridis and K. Norvag, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, pp. 1–26, 2013.

[4] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: A survey," *SIGMOD Rec.*, vol. 40, pp. 11–20, Jan. 2012.

[5] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 975–986, ACM, 2010.

[6] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, (San Francisco, CA, USA), pp. 27–40, Morgan Kaufmann Publishers Inc., 1992.

[7] F. Atta, S. Viglas, and S. Niazi, "Sand join - a skew handling join algorithm for google's mapreduce framework," in *Multitopic Conference (INMIC), 2011 IEEE 14th International*, pp. 170–175, Dec 2011.

[8] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 949–960, ACM, 2011.

[9] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[10] T. Lee, K. Kim, and H.-J. Kim, "Join processing using bloom filter in mapreduce," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, (New York, NY, USA), pp. 100–105, ACM, 2012.

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

[12] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using mapreduce," *Proc. VLDB Endow.*, vol. 5, pp. 1184–1195, July 2012.

[13] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, (New York, NY, USA), pp. 99–110, ACM, 2010.

[14] J. Myung, J. Yeon, and S.-g. Lee, "Sparql basic graph pattern processing with iterative mapreduce," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, (New York, NY, USA), pp. 6:1–6:6, ACM, 2010.

[15] C. B. Walton, A. G. Dale, and R. M. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins," in *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, (San Francisco, CA, USA), pp. 537–548, Morgan Kaufmann Publishers Inc., 1991.

[16] M. Kitsuregawa and Y. Ogawa, "Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc)," in *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, (San Francisco, CA, USA), pp. 210–221, Morgan Kaufmann Publishers Inc., 1990.

[17] K. A. Hua and C. Lee, "Handling data skew in multiprocessor database computers using partition tuning," in *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, (San Francisco, CA, USA), pp. 525–535, Morgan Kaufmann Publishers Inc., 1991.

[18] L. Harada and M. Kitsuregawa, "Dynamic join product skew handling for hash-joins in shared-nothing database systems," in *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*, pp. 246–255, World Scientific Press, 1995.

[19] A. Shatdal and J. F. Naughton, "Using shared virtual memory for parallel join processing," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, (New York, NY, USA), pp. 119–128, ACM, 1993.

[20] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, (Washington, DC, USA), pp. 522–533, IEEE Computer Society, 2012.

[21] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 25–36, ACM, 2012.

[22] R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in a relational data base system," in *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, SIGMOD '78, (New York, NY, USA), pp. 169–180, ACM, 1978.

[23] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM JOURNAL ON APPLIED MATHEMATICS*, vol. 17, no. 2, pp. 416–429, 1969.

[24] H. Lu and K. Tan, "Load-balanced join processing in shared-nothing systems," *Journal of Parallel and Distributed Computing*, vol. 23, no. 3, pp. 382 – 398, 1994.

[25] C. Hahn and S. Warren, *Extended Edited Synoptic Cloud Reports from Ships and Land Stations Over the Globe, 1952-1996*. Environmental Sciences Division, Office of Biological and Environmental Research, U.S. Department of Energy, 1999.

[26] M. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Physics*, vol. 46, pp. 323–351, Sept. 2005.

[27] J. D. Gibbons, *Nonparametric Methods for Quantitative Analysis (3rd Ed.)*. Syracuse, NY, USA: American Sciences Press, 1997.

[28] M. Muralikrishna and D. J. DeWitt, "Equi-depth multidimensional histograms," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, (New York, NY, USA), pp. 28–36, ACM, 1988.

[29] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "An evaluation of non-equijoin algorithms," in *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, (San Francisco, CA, USA), pp. 443–452, Morgan Kaufmann Publishers Inc., 1991.

[30] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Trans. Database Syst.*, vol. 9, pp. 38–71, Mar. 1984.

[31] R. Z. Albert, *Statistical Mechanics of Complex Networks*. PhD thesis, Notre Dame, IN, USA, 2001. AAI3000268.

[32] "Sparql (query language for rdf). http://www.w3.org/tr/rdf-sparql-query/."

[33] "Rdf (resource description framework). http://www.w3.org/rdf/."

[34] T. H. Cormen, *Introduction to algorithms*. MIT Press, 2001.

[35] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[36] R. R. Amossen and R. Pagh, "Faster join-projects and sparse matrix multiplications," in *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, (New York, NY, USA), pp. 121–126, ACM, 2009.

[37] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, (New York, NY, USA), pp. 47–57, ACM, 1984.

[38] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semant.*, vol. 3, pp. 158–182, Oct. 2005.

[39] "Owl (web ontology language). http://www.w3.org/tr/owl-features/."

[40] "Apache jena. http://jena.apache.org/."

[41] "Apache hbase. https://hbase.apache.org/."

[42] T. Neumann and G. Weikum, "x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases," *Proc. VLDB Endow.*, vol. 3, pp. 256–263, Sept. 2010.

[43] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic web data management," *Proc. VLDB Endow.*, vol. 1, pp. 1008–1019, Aug. 2008.

[44] J. Dean and S. Ghemawat, "Mapreduce: A flexible data processing tool," *Commun. ACM*, vol. 53, pp. 72–77, Jan. 2010.

[45] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung, "Spider: a system for scalable, parallel / distributed evaluation of large-scale rdf

data.," in *CIKM* (D. W.-L. Cheung, I.-Y. Song, W. W. Chu, X. Hu, and J. J. Lin, eds.), pp. 2087–2088, ACM, 2009.

[46] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, (Washington, DC, USA), pp. 277–284, IEEE Computer Society, 2008.

[47] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2010.

[48] J. D. U. Anand Rajaraman, *Mining of Massive Datasets*. Cambridge University Press, 2012.

[49] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *ICDE*, pp. 231–242, 2011.

[50] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, (Washington, DC, USA), pp. 229–238, IEEE Computer Society, 2009.

[51] I. v. Milentijević, I. Z. Milovanović, E. I. Milovanović, M. B. Tošić, and M. K. Stojčev, "Two-level pipelined systolic arrays for matrix-vector multiplication," *J. Syst. Archit.*, vol. 44, pp. 383–387, Feb. 1998.

[52] J. Myung and S.-G. Lee, "Exploiting inter-operation parallelism for matrix chain multiplication using mapreduce," *J. Supercomput.*, vol. 66, pp. 594–609, Oct. 2013.

[53] J. Norstad, "A mapreduce algorithm for matrix multiplication, http://homepage.mac.com/j.norstad/matrix-multiply/index.html," 2009.

[54] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, (Washington, DC, USA), pp. 721–726, IEEE Computer Society, 2010.

[55] "Stanford large network dataset collection, http://snap.stanford.edu/data/index.html."

[56] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, pp. 1626–1629, Aug. 2009.

[57] "Apache giraph, http://incubator.apache.org/giraph/."

[58] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[59] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[60] M. F. Pace, "Bsp vs mapreduce," *CoRR*, vol. abs/1203.2081, 2012.

Index

초 록

조인은 대부분의 데이터 분석 작업에서 사용되는 중요한 연산이지만, 맵리듀스에서 직접적으로 지원되지는 않는다. 이것은 맵리듀스가 기본적으로 하나의 입력 데이터만을 다루도록 설계되었고, 또한 맵리듀스의 '키-일치 (key-equality)' 방식의 데이터 처리 흐름이 다양한 조인 조건을 수용하기 어려웠기 때문이다. 이에 따라, 맵리듀스에서의 조인에 대한 수 많은 연구 결과와 알고리즘들이 발표되었으며, 지금도 활발하게 연구가 진행중이다.

여타의 비공유(shared-nothing) 시스템에서와 마찬가지로, 맵리듀스에서의 조인 알고리즘이 갖는 중요한 문제 가운데 하나는 데이터의 불균형(skew)에 대한 처리 문제이다. 이것은 가장 늦게 종료하는 계산 노드가 전체 수행시간을 결정하는 비공유 시스템의 특징 때문이다. 이를 해결하기 위해서 지금까지 범위 분할 (range partitioning) 방식과 임의 분할 (random partitioning) 방식이 사용되어 왔으나, 이들은 각각 특정한 상황에서 약점을 가지고 있다.

따라서 이 논문에서는 이전 방식들의 문제점을 살펴보고, 이에 대응할 수 있는 새로운 불균형 처리 기법을 제시한다. 제안된 기법은 다차원 범위 분할 (Multi-Dimensional Range Partitioning) 방식으로 명명했으며, 이 논문에서는 다양한 실험을 통해 제안 방식이 기존 방식보다 좋은 성능을 보임을 입증한다. 보다 구체적으로, 1) 제안된 방식은 기존의 범위 분할 방식에 비하여 조인 결과의 크기에 대한 고려를 가능하게 한다. 이는 개별적 입력에서 불균형이 없더라도 결과에서 불균형이 발생하는 경우를 처리하는데 도움이 된다. 2) 제안 방식은 기존의 임의

분할 방식과 비교하여, 조인 조건을 사전에 활용하여 발생 가능성이 없는 입력 쌍에 대해 여과를 가능하게 한다. 이는 불필요한 입력의 중복 전송을 방지하여 네트워크 비용을 감소시키고, 불균형이 존재하지 않을 경우에도 좋은 성능을 발휘하도록 해준다.

제안 방식은 기존의 맵리듀스 프레임워크에 대한 수정 없이, 세타 조인 및 멀티웨이 조인과 같은 진보된 연산에 대해서도 처리가 가능하다. 우리는 여러 데이터 셋에서 수행된 다양한 실험 결과를 통해 제안 방식이 기존 방식보다 효과적임을 보인다.

주요어 : 병렬 조인, 불균형 데이터, 다차원 범위 분할, 맵리듀스
학번 : 2007-20973